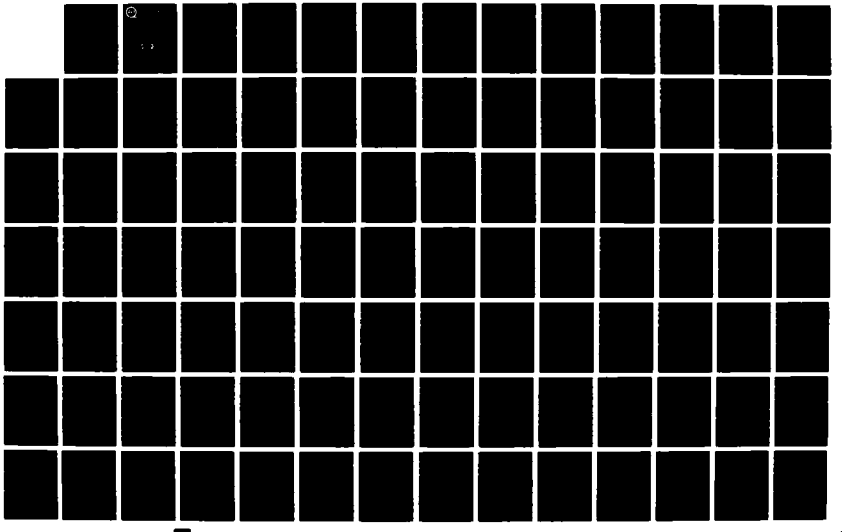
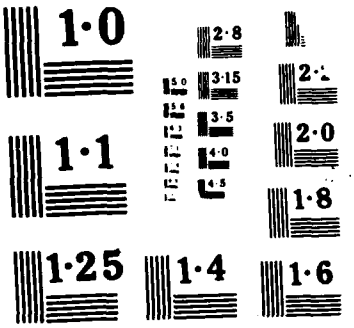
AD-A191 204 A TIMING EVALUATOR FOR C PROGRAMS GENERATED BY THE MODEL SYSTEM(U) RENSSELAER POLYTECHNIC INST TROY NY DEPT OF COMPUTER SCIENCE M SRINIVASAN ET AL. DEC 87 RPI-TR-87-29 N00014-86-K-0442 1/2

UNCLASSIFIED F/G 12/5 NL




1·0
1·1
1·25
1·4
1·6
2·8
3·15
3·5
2·0
1·8

DTIC FILE COPY

Department of Computer Science

Technical Report

AD-A191 204

A Timing Evaluator for C Programs Generated by the Model System

Mahesh Srinivasan

and

Boleslaw Szymanski

DTIC
ELECTE
FEB 19 1988
S D

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Report No. 87-29

December, 1987

88 1 20 029

A TIMING EVALUATOR FOR C PROGRAMS GENERATED BY THE MODEL SYSTEM

by

Mahesh Srinivasan

Project Advisor

Prof. Boleslaw Szymanski

Submitted to Information System Program
Office of Naval Research
Under Contract N00014-86-K-0442

Technical Report

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By per ltr.
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

CONTENTS

Page

LIST OF TABLES iii

ABSTRACT iv

1. INTRODUCTION 1
   - 1.1 The MODEL System 1
   - 1.2 Using the MODEL system 3
   - 1.3 The Timing Evaluator 4

2. USING THE MODEL TIMING EVALUATOR 6
   - 2.1 Introduction 6
   - 2.2 The input C program 7
   - 2.3 The instruction timing data file 10
   - 2.4 The function timing data file 14
   - 2.5 The Timing Report 17
   - 2.6 Parameters to the MTE 20

3. MTE INTERNALS 22
   - 3.1 Introduction 22
   - 3.2 The Lexical Analyzer 23
   - 3.3 The Parser 24
   - 3.4 Symbol Tables and Types 25
   - 3.5 Code Generation 27
   - 3.6 Timing Evaluation 29
   - 3.7 Timing Report generation 31

4. LIMITATIONS OF THE MTE 33
   - 4.1 Introduction 33
   - 4.2 The Target Machine 33
   - 4.3 MTE Internal Inaccuracies 34
   - 4.4 Future Modifications 35

APPENDIX 1 36

APPENDIX 2 47

APPENDIX 3 56

LIST OF TABLES

Page

Table 2.1 Instruction Set 11

Table 2.2 Addressing Modes 12

Table 2.3 An Example Timing Report 18

Table 2.4 Parameter file 21

ABSTRACT

A timing analyzer for the C programs generated by the MODEL compiler has been developed. Timing analysis is carried out before the input programs are executed on the target machine. The overall module delay and the relative delays between the module input and output events are reported.

CHAPTER 1

INTRODUCTION

1.1 The MODEL System

The MODEL system was developed to address the problems involved in the development and maintenance of real time software for embedded computer systems. The primary goal of the MODEL system is to ease the task of developing and maintaining real time software by using a nonprocedural language approach. The secondary goal is to provide automated support to assist programmers in meeting the time constraints imposed by real time software systems.

A real time software system can be defined as one which controls an environment by receiving data, processing it and taking action or returning results quickly enough to affect the functioning of the environment at that time. Real time applications are usually connected to embedded computer systems where software controls the operation of a host system such as an aircraft, ship, or factory. Examples of such applications are flight control systems, radar tracking systems, industrial process control systems, etc.

Some of the major problems involved in the development and maintenance of real time software are :

1. The software is typically large and complex. This makes the software development process expensive since it is man power intensive.
2. Real time requirements put high demands on both reliability and performance. Both the time constraint on program response time and program correctness are of vital importance in real time applications.
3. During the system life cycle, change is continuous due to the evolution of environment and technology. To keep the system up to date with the state of the art technology, it is essential that maintenance of the software system be easy and convenient.

Another major concern for real time systems is writing concurrent programs. This is due to

the asynchronous parallelism characteristic of these systems.

In order to handle the problem of size and complexity, automated supports have to be provided as much as possible. A nonprocedural language, MODEL, with the capability of automatic code generation, providing high level operations, and accomodating concurrent systems, has been developed. In order to check that the timing constraints imposed on real time systems are met, a **MODEL Timing Evaluator** has been developed in this research to provide information about module (program component) delays between comunications with other modules or the outside world.

The MODEL approach aids or automates the following portions of the software development and maintenance process:

1. High level source code generation from specification - a very high level, nonprocedural language (MODEL) is provided for writing the software specifications. The **MODEL compiler** uses this to generate code in a high level programming language (C, ADA, Fortran, or PL/1).
2. Synchronization and communication channels between modules executing in parallel. These are established by the **Configurator** using a specification from the user written in the Configuration Specification Language (CSL).
3. Simulation - An executable model of the system that runs on the host computer is produced as a result of using the MODEL compiler and Configurator. This model can be used for testing, debugging & performance study purposes.
4. Documentation - A number of reports are generated automatically. The following is a partial list: the system design & structure, individual program listing, generated C (or Fortran, ADA, or PL/1) code listing and timing reports.
5. Early timing performance analysis - Normally, the timing study can be done only after programs in target machine code have been produced and executed. Instead, with the help of the timing evaluator, performance analysis can be done when an individual

module has been specified, even on a host other than the target machine.

1.2 Using the MODEL system

The MODEL software development support system consists of three parts - the MODEL language and compiler, the Configurator, and the MODEL Timing Evaluator (MTE). It is the Timing Evaluator that has been developed by the author and is the main subject of this report. The MODEL compiler can at present generate code in any of the four high level programming languages - C, ADA, Fortran, or PL/1. However, the MTE that has been developed is useful only for the case in which C code is generated by the MODEL compiler. Henceforth, any reference made to the object programs produced by the MODEL compiler is a reference to the C programs it generates.

The real time software development process begins with the availability of the software system requirement, which usually consists of three parts :

1. Functional requirements - defining the functions and subfunctions of the system.
2. Performance requirements - time constraints for time-critical performance of the system.
3. Interface with the environment - the layout of the data communicated with the environment.

The software designer begins by dividing the system functions into into software modules and data files. A function may be carried out by one one or more modules, or several related functions may be combined into one module. The relationship and communications between modules are also defined at this point. The modules are in skeletal form, with only the external data structures outlined as files. The designer can now use the Configurator to verify global system consistency and completeness.

Next, the designer composes the module specifications independently for each module in the MODEL language. The MODEL compiler processes each module separately, performing completeness and consistency checks within each module, and in the absence of errors, generates a C program to perform the task of that module. The user can now employ

the Timing Evaluator on each generated C program to verify if the time constraints associated with the corresponding module are satisfied. The Timing Evaluator produces a Timing Report for each module that provides information on time delays between instances of input and/or output in the module. The user has to provide certain timing data of the target machine to the Timing Evaluator for it to generate the Timing Report.

The designer may also have to check if global time constraints are met by adding individual module delays in a path of the configuration to obtain the overall delays between critical events involving multiple modules. If some of these constraints are not satisfied, the designer may have to modify the configuration of the entire system by partitioning some modules to obtain a greater degree of parallelism.

Once all the modules have been satisfactorily processed by the MODEL compiler and the timing evaluator, the designer uses the Configurator to synthesize all the system components (modules and data files) into an integrated system. The user composes the system by specifying a configuration of modules and files in the Configuration Specification Language that is the input to the Configurator. It then schedules individual modules, synchronizes modules that will execute in parallel, sets up communication channels between modules, and generates command language procedures that will run the C programs with maximum concurrency in the host computer's multiprogramming / multiprocessing environment.

Finally, the system can be executed and tested on the host machine, code can be generated for the target machine using cross compilers, and the real time software system can be tested on the target machine.

1.3 The Timing Evaluator

The MODEL Timing Evaluator (MTE), which is the subject of this report , has been developed to assist MODEL users in determining whether their real time software meets the initial timing requirements without having to actually test run the software on the target

machine. The MTE can be integrated with the MODEL compiler and can be activated as an option of the compiler. Timing information about individual modules can be obtained as soon as the compiler determines that the module specification is error free and generates a C program to implement a particular module.

Briefly, the MTE operates as follows:

1. Reads in timing data of the target machine from a file.
2. Reads in the MODEL compiler generated C program, translating it into machine level instructions and calculating execution times for these instructions. Input/Output operations in the program are remembered for use at the end in preparing the timing report.
3. A Timing Report is printed, giving the delays caused by I/O operations and the worst case time delays between different I/O events. The worst case total time of program execution is also reported.

The next three chapters describe the MTE in detail. Chapter 2 is a user manual for the MTE , chapter 3 describes the internal design and implementation of the MTE and chapter 4 concludes by discussing the limitations of the MODEL Timing Evaluator.

CHAPTER 2

USING THE MODEL TIMING EVALUATOR

2.1 Introduction

The MTE was developed on the Sequent Balance 21000 running the DYNIX operating system which is a variation of the UNIX operating system. Though the MTE can be used on any system the example commands that follow apply to all UNIX systems only. The MODEL Timing Evaluator will be referred to by the abbreviation MTE.

The MTE source program is made up of 4 files :

1. parameters.h
2. lex.src
3. yacc.src
4. aux.c

All four files should be in the same directory. To construct an executable object file on a UNIX system, the following commands should be used:

hostname% lex lex.src

hostname% yacc yacc.src

hostname% cc -o mte y.tab.c -lm

The first command produces a C file **lex.yy.c** while the second command produces the file **y.tab.c**. The final compilation produces the executable file, **mte**. The last two commands both produce warning messages which can be ignored.

Having obtained the executable MTE program, it can now be run to perform a timing analysis of a C program produced by the MODEL compiler. The MTE reads the C program from standard input, takes the names of 2 data files as arguments, and prints the timing report on standard output. An example :

hostname% cc -E -C *program.c* | mte *idata fdata* > *report*

where **idata** is a file that contains the instruction timing data for the target machine, **fdata** is

a file that contains the function timing data and **program.c** has the C program to be analysed. The timing report is put in the file **report** and can be printed on the system printer. Note that the C preprocessor has to be used on the C program for macro substitutions while preserving comments in the program since these comments contain useful information for MTE. The rest of this chapter is devoted to the explanation of the syntax/semantics/restrictions on the contents of the above files.

2.2 The input C program

The MTE takes advantage of the fact that the MODEL compiler uses only some of the features of the C programming language in its automatic generation of C programs from the input MODEL language specification. Therefore, though the MTE is primarily designed to be used with the MODEL system, it can actually be used to perform timing analysis on any C program that satisfies the following constraints/assumptions/requirements :

1. The C program is correct, syntactically and semantically. Since the input to MTE is generated by another program, namely the MODEL compiler, no errors are expected. The MTE does not perform extensive semantic checking and its behaviour is unpredictable if faced with semantic errors.
2. The program has exactly one function called **main** and this function is not recursive. Moreover, the times of all the functions that are called from **main()** are present in the function timing data file. This applies even to functions whose bodies are present in the same file as the MTE performs timing analysis of only the **main()** function.
3. It is assumed that certain legal C statements will not be present in the input program. These are the **entry, fortran, & asm** statements.
4. The MODEL compiler also generates comments in the C program it produces. Some of these comments improve readability of the program while some comments convey information to the MTE that is necessary to calculate time delays. Comments intended for the MTE are distinguished from other comments by having the character string **MTE**

at the beginning of the comment. There are exactly two cases in which these comments should be used :

a. Loop ranges - every loop in the **main()** function should be preceded by a comment that has a constant integer which will be interpreted as the number of iterations the loop will perform before exiting. If this number is not known, then the maximum number of iterations should be provided. This enables the MTE to evaluate the worst case delay caused by the loop. A missing comment will be treated as an error and the MTE will halt without producing any timing report.

Eg;

```
/* MTE  5 */
   for (i = 0; i < j; i++)
     {
        .....
     }
```

This will cause MTE to assume that the loop is executed 5 times.

b. Filenames for I/O operations - the timing report produced by MTE gives the time delays between input/output operations in the C program. In order to make this report more readable, the name of the file on which an I/O operation is being performed can be given as a comment before the call to the I/O function. If there is a comment, MTE uses the given filename in its timing report, otherwise it refers to the I/O operation by the linenumber in the C program on which it appears.

Eg;

```
/* MTE OUTT */ write(..,..,..);
```

will cause MTE to refer to this I/O operation in the timing report as "write OUTT".

5. All variables appearing in the **main()** function are declared within the same function. Those that have been declared elsewhere will be considered to be of type **int** & this

assumption might lead to inaccuracies in the calculation of timing data. All variable declarations are before the first executable line in the **main** function. There should be no declarations within blocks inside the main function.

6. The only legal variable types recognized by the MTE are the basic scalar types of the C language, arrays of objects of legal type & structures and unions of objects of legal type. Please note that the pointer type is not recognized and its appearance in the C program will cause an error. It is also assumed that there is no **typedef** statement within the main function and no usage of a defined type within the main function. Furthermore, typecasts can contain only basic scalar types and not more complex types such as structures or arrays.

7. If **a** & **b** are structures (or unions) of the same type, there is no statement of the form

   a = b ;

   Also, structures are not passed as parameters to, or returned by, functions.

8. There is a provision to express function timings as an arithmetic expression of the number of arguments in a call to the function, and/or the constant integer that is passed as a parameter to the function, and/or the dimension of a string that is passed as a parameter. In the last two cases, the actual calls to the function in the program should pass, respectively, constant integers and one dimensional character arrays as parameters. Anything else would cause an error as the MTE would be unable to evaluate the function delay. If a one dimensional character array is going to be used as a parameter to a function whose timing expression requires the size(dimension) of the array, then the declaration for the character array should contain the size explicitly as a constant integer.

   Eg; char a[20], b[][10], c[];

   Both **a** and ***b[i]*** can be used as string parameters to functions that need the string size, but not **c**.

9. There is no dynamic memory allocation in the main function. This is actually implied by the earlier constraint that there is no use of pointers.

Though the above list may seem very restrictive, it is fully satisfied by the C programs generated by the MODEL compiler and greatly simplifies the design and implementation of the MTE.

2.3 The instruction timing data file

The MTE works by compiling the input C program to machine level instructions that can be found on most general purpose computers and adding the individual instruction execution times to arrive at the total time delay. In order to function, the MTE needs the execution times of certain instructions for the taget machine in question. This data file has to be changed each time the user wants a timing analysis for a different target computer.

The syntax of the instruction timing data file,**idata**, is simple - it should be a file containing 279 numbers, integer or floating point. These numbers correspond to the execution times of 279 different instructions, all times being in microseconds. The assumed instruction set and addressing modes are listed in Tables 2.1 and 2.2. To help the user to prepare this file, there is another file, **idata.template** that has the names of the instructions and useful explanations enclosed within comments("/*" & "*/") along with the corresponding execution times. The file **idata** can be obtained by simply running the C preprocessor on the file **idata.template** to remove comments. The template file for the Sequent Balance 21000 is listed in Appendix 1. Obviously, the order of the instructions in the file is fixed and should not be changed.

A few basic and unavoidable assumptions have been made about the target machine:

1. The target computer has floating point hardware to support the floating point instructions found in Appendix 1. Most modern day general purpose computers satisfy this requirement.

Table 2.1
Instruction Set

Instruction	Explanation
RET	Return from function call
BR	Unconditional branch
BRcond	Conditional branch
Scond	Conditional set
MOD	Modulus
AND	Bitwise and
OR	Bitwise inclusive or
XOR	Bitwise exclusive or
COMP	One's complement
ASH	Arithmetic shift
ITOF	Convert integer to floating
FTOI	Truncate floating to integer
ADDR	Form the address
MOV	Move
ADD	Addition
SUB	Subtraction
MULT	Multiplication
DIVD	Division
NEG	Negation
CMP	Compare

2. The computer has the four listed addressing modes - in fact, most computers possess a much richer set of addressing modes but to be on the safe side, only these four have been assumed.
3. The size of the C data type **int** is less than or equal to the size of a word of memory. In most machines, the sizes are the same. "Integer" instructions in the template file refer to data that is the size of the C data type **int** and "Floating" instructions refer to the type **float**. Where no type is mentioned, it is implied that only one type is possible & that is "Integer".
4. The size of a pointer on the machine is the same as the size of the **int** data type. If not, the time delays reported could be inaccurate & on the optimistic side.

There are two instructions in the listed set that require some explanation. If these instructions are not present on the target computer, alternative sequences of instructions

Table 2.2
Addressing Modes

Mode	Explanation
IMDTE	Immediate - constant operand
REG	Register - operand in a register
ABS	Absolute - address of the operand
REG_REL	Register relative - the address is offset + register

can be used to calculate execution times :

1. **ADDR** forms the address of the source operand and places it in the destination operand.

   ADDR ABS,*dest*

   is equivalent to

   MOVi IMDTE,*dest*

   and

   ADDR REG_REL,*dest*

   is equivalent to

   MOVi REG,*dest*

   ADDi IMDTE,*dest*

2. **Scond** sets or clears its operand depending on whether ***cond*** is true or false.

   S*cond dest*

   is equivalent to

   ```
   BRcond  true
   MOVi    IMDTE,dest
   BR      next
   true:  MOVi    IMDTE,dest
   next:  .....
   ```

The worst case timing should be used for the above instruction sequence.

Finally, some hints on how to find the individual instruction times. It is assumed that the input C program executes on the target machine in a dedicated mode with the entire program in main memory (ie) the MTE does not account for delays caused by involuntary context switching or paging of virtual memory. One way to find the instruction timings is from the manufacturer's data - but this is often unreliable and overly optimistic; so this method has not been explored. The other method is to directly experiment on the target machine to obtain the instruction timings and is described here.

Since individual machine instructions take only a few microseconds or even less to execute, and since it is difficult to measure such time intervals accurately even on a computer, each instruction can be executed in a loop a few million times and the elapsed time noted. Next, the empty loop (without any instruction in its body) can be executed the same number of times and the elapsed time for this can be stored. From this, by simple subtraction and division, the execution time for a single machine instruction can be found. A few things to note :

1. The **asm** statement can be used to directly insert an assembly language instruction in a C program. This can simplify the programming effort involved.
2. If the target machine architecture is such that instructions are prefetched from memory, the measured instruction execution time could be greatly exaggerated (sometimes by 100%) if this program segment is used :

   for (i=0; i < 1E6; i++) asm("*instr*");

   This is because the repeated branching nullifies the effect of prefetching and is not typical of normal programs. Instead the following segment could be used to obtain more accurate timing data :

```
for (i=0; i < 1E4; i++)
  { asm("instr");
```

```
asm("instr");
    ...
    ...
  100 times
}
```

This ensures that prefetching is in effect for 97% - 98% of the instructions executed within the loop.

3. Elapsed time intervals can be measured on UNIX systems using the system calls **setitimer** & **getitimer**. These calls provide an interval timer that can measure virtual process time (ie) the timer decrements only when the process is executing.
4. The above methods can be used to find the timings for most instructions. However, branch instructions require a slightly different approach. Instead of executing the instruction a fixed number of times and measuring elapsed time, branch instructions can be executed for a fixed amount of time and then interrupted. A counter can keep track of the number of iterations the infinite loop (2 branch instructions jumping to each other) goes through.

2.4 The function timing data file

Whenever the MTE encounters a function call in the **main()** function of the input C program, it attempts to find the time of execution of the function from the data in the function timing data file, **fdata**, to evaluate the time delay caused by the function. The MTE does not differentiate between C library functions, user defined functions and system calls. Every possible function that can be called from the **main()** function in the input C program should be present in the function timing data file and have a valid timing expression associated with it.

The **fdata** file should have one function to each line, with the name of the function at the beginning of the line, followed by some whitespace, followed by the timing

expression for the function, the syntax of which is described below. Since the execution times of some functions, like string manipulation functions & I/O functions, can depend on the parameters that are passed to them, expressions, rather than constant numbers, are used to indicate their execution times. The timing expression for a function is evaluated and reduced to a numerical time for each call to the function in the C program. The syntax of the timing expression for a function can be described by the following context free grammar:

expression --> terminal
| expression + expression
| expression - expression
| expression * expression

terminal --> constant_number
| a*n*
| A*n*

where **constant_number** can be an unsigned integer or floating point number and *n* is an unsigned integer ≥ 0. Note that no parantheses are allowed and the usual priorities hold :

plus = minus < times

Only decimal format is allowed for **constant_number** - the exponential format is not recognized. All times should be in microseconds. The meaning of a*n*(or A*n*) is as follows :

1. a0 is the number of arguments passed to the function in a call to the function.
2. a*n*, n ≥ 0, is the value associated with the *n*th argument passed to the function in a call to the function. The value associated with an argument is defined below :
   a. If the argument is a constant integer, then its value is equal to the constant integer.

b. If the argument is a one dimensional character array or a constant string, its value is equal to the dimension of the array or length of the string.

c. Otherwise the value of the argument is undefined.

Similar to the instruction timing data template file, there is a file **fdata.template** to assist users in preparing the **fdata** file. The **fdata.template** file for the Sequent Balance 21000 computer is listed in Appendix 1. This file is not complete since the MODEL compiler is being rewritten at the time of this report and a complete list of functions being used is not available.

Function times should indicate the delay from the beginning of execution of the **branch to subroutine** to the beginning of execution of the next instruction in the calling function. This is why there is no entry for a **branch to subroutine** instruction in the instruction timing data file - it is always included in the function time. In the case of functions used for communicating with other concurrently executing modules, delays may be caused by having to wait for the other module(s) to be ready for communication. These delays are not accounted for in the timing expressions for these functions since they are unpredictable. Only the sending/receiving time is considered and not the waiting time.

Function times for a target machine can be determined in a manner similar to instruction times - by executing the functions repeatedly and measuring the elapsed time. In the cases of functions like **strcpy** and **read**, whose times are dependent on the parameters passed to them, their execution times can be determined for several different parameter values and a first degree polynomial with the parameter values as independent variables can be found - this polynomial would be the timing expression for that particular function.

Lastly, if the timing expression for a function involves one or more values associated with its arguments, then these values should be defined in every call to the function within the **main()** function of the C program being analysed. Otherwise, the MTE would be unable to evaluate the time delay caused by the function and it will halt after printing an error

message. Moreover, if the MTE is to know the dimensions of character arrays, these arrays should be declared within the **main()** function and the sizes declared should be constant integers.

2.5 The Timing Report

The timing report is the output of the MTE and gives the total execution time of the program and the time delays between critical events in the program. These critical events include I/O operations and message transfers from/to concurrently executing programs (modules in the overall MODEL specification). An example MODEL program and the C program generated for it by the MODEL compiler are listed in Appendix 2. The timing report produced by the MTE for this C program is given in Table 2.3.

The general philosophy behind the timing report produced by the MTE is to always consider the worst case time delays - when it is faced with conditional structures like the **if-then-else** statement or the **switch** statement, it has no way of predicting which path will be chosen during program execution and so the MTE assumes that the longest path is always chosen. This is due to the fact that if the software system satisfies the time constraints in the worst case, it will always satisfy those constraints. When the MTE processes loop statements in the C program, it expects to see a **loop range** before each loop as described in Section 2.2. This integer gives the number of iterations the loop will go through or, if that is unknown, the maximum number of iterations for the loop. Using this **loop range**, the MTE is able to determine the worst case time delay caused by the loop.

The Timing Report gives the time delay from every critical event to every other critical event for which there is a possible path (flow of control) from the 1st event. If a critical event occurs within a loop, then the time from one occurrence of the event to the next is also reported. A critical event in the program is simply a call to one of a set of special functions - for more details refer to the next section. Normally this set of special functions would contain I/O and communication (with other modules) functions but there is no

Table 2.3
An Example Timing Report

```
***** MODEL TIMING EVALUATOR

***** TIMING REPORT

    LEGEND
    ******

No.    NAME                          TIME
---    ----                          ----
  0    PROGRAM BEGIN                 0.000 millisecs
  1    read            ACMINS        2.259 millisecs
  2    write           ACMOUTT       2.360 millisecs
  3    PROGRAM END                   0.000 millisecs

    All times in MILLISECONDS
All times FROM beginning of one event TO beginning of next event
```

	<- TO ->			
From	0	1	2	3
0	**	18.134	7805.761	7808.427
1	**	**	7787.627	7790.293
2	**	**	**	2.666
3	**	**	**	**

```
Total Execution time = 7808.427 millisecs

***** Exiting MODEL timing evaluator
```

inherent limitation in the MTE as to the nature of these functions. In addition to the above, there are two predefined critical events - the physical program beginning and program end (ie) the 0th line in the **main**() function and the (last+1)th line of the function.

The **LEGEND** in the timing report lists all the critical events (special function name followed by a filename if present, else its line number in the C program), their time of duration (delays caused by the events), and their serial numbers by which they are referred to in the rest of the timing report. After the **LEGEND**, the report gives the delays between critical events in a tabular form.The times are all in milliseconds, and the delay from the beginning of one event to the beginning of another event is the table entry corresponding to the row for the first event & the column for the second event. If there is no possible flow of control from one event to another, the corresponding table entry is "***". The time delays reported in the table are to be interpreted as follows :

1. If both events are not inside any loop, then the delay is just the obvious time difference between the 1st event and the 2nd.
2. If the 1st event is within a loop and the 2nd event is not in that loop, then the delay is the time between the last occurrence of the 1st event and the occurrence of the 2nd.
3. If the 2nd event is within a loop and the 1st event is outside that loop, then the delay is the time between the 1st event and the first occurrence of the 2nd event.
4. If both the events are inside the same loop, the delay is the shortest possible time between an occurrence of the 1st event and an occurrence of the 2nd.

These rules can be applied recursively to critical events occurring within nested loops. A few things to note:

1. Every possible path that can be followed by the executing program is considered in reporting delays between critical events even though only the longest path between any two events is used for evaluating the overall delay between any two events. In other words, by always being pessimistic in calculating time delays, some of the worst case delays may turn out to be mutually exclusive (ie) they cannot simultaneously occur in a single execution of the program.

2. When two critical events occur within the same loop, the shortest possible time is reported because it is impossible for the MTE to predict which path the program will follow in which iteration of the loop.

Finally, the reported time delays are by no means accurate - they are conservative estimates that serve the purpose of the MTE in helping to schedule concurrently executing modules in an optimal way to obtain minimum execution time (or respose time in the case of real time software) for the whole software system. More about the limitations of the MTE can be found in chapter 4.

2.6 Parameters to the MTE

The file **parameters.h** is used to convey certain operational parameters to the MTE and may need alteration when the MTE is ported to a new system or when internal data structures overflow. Since the file **parameters.h** is part of the source code of the MTE program and not a data file, the MTE executable image has to be reconstructed each time the file is altered. A listing of the current version of the file is shown in Table 2.4.

Names of functions that are to be treated as critical events should be placed in the array **special_funcs**. When an internal overflow occurs, MTE prints a message telling the user that an internal error has occurred and the name of the parameter that needs to be changed. Error messages are also generated when the MTE detects any syntax errors in the input C program / the two data files or if there are any violations of some of the constraints listed in section 2.2. All errors cause the MTE to halt with a return code of 1 and no Timing Report will be generated.

Table 2.4
Parameter file

```
/********************** FILE "parameters.h" ************************
17th Nov. '87                                   Mahesh K. Srinivasan

        This file contains parameters to the MODEL Timing Evaluator
program that need to be altered when porting to a different system or
performing timing analysis on certain types of C programs or altering
the code generation part of the MODEL compiler.
NOTE : After any alteration is made, the MTE program will have to be
         recompiled. This file is "#include"d in file "yacc.src".
*******************************************************************/
/* The tag at the beginning of comments intended for the MODEL Timing
   evaluator in the input C program is the string below, without the
   quotes.                                                          */
#define COMMENTAG "MTE"
/* Initializing the debug flag to a value > 0 causes the MTE to print
   the machine language instructions it generates to calculate time
   delays.                                                          */
int debug_flag = 0;

/*  The following parameters determine sizes of various internal data
   structures of the MTE. If there is an internal overflow, the MTE
   issues an error message that gives the name of the parameter whose
   definition has to be increased. After recompilation, the MTE can be
   used for analysis of the program that caused the overflow.       */
typedef short type_type;
#define MAXD        9
#define MAXSTRUCS   10
#define MAXIFNEST   20
#define MAXLOOPNEST 20
#define MAXARGS 10
#define MAXDEPTH 5
#define MAXFUNCS 50
#define MAXLEN 50
#define MAXSSTACK 10

/*  Every string in the array below is interpreted as the name of an
   I/O or communication function. Adding the name of a function to
   this array will cause the MTE to treat every call to the function
   as a critical event that will figure in the Timing Report.       */
 char *special_funcs[] = {"read","write"};

/*  NCOLS will determine the number of columns per table printed in
   the Timing Report. MAXNAME is the numer of significant characters
   for identifier names, and N_BUCKS is the number of buckets used in
   the hash tables. Can be increased to improve speed.              */
#define NCOLS 6
#define MAXNAME 20
#define N_BUCKS 10
/*                  ****************                                */
```

CHAPTER 3

MTE INTERNALS

3.1 Introduction

The basic algorithm of the MTE is to compile the input C program down to machine language instructions, and using the timing data provided by the user, calculate time delays. The MTE was developed on the Sequent Balance 21000 and implemented in C. It tries to mimic the Sequent C compiler without using any of the specialized instructions of the machine's NS3200 processors - a compromise between accuracy and portability.

The code that the MTE generates, in order to calculate the time delays, is not intended to run on any machine; this made the task of designing the MTE a lot simpler than any actual C compiler. Moreover, since the input C programs are generated by another program (the MODEL compiler), it is known beforehand that not all of the features of the C programming language will be used and the MTE has been designed to take advantage of this fact. For more about the features of the C language not used by the MODEL compiler, refer to section 2.2.

The MTE is organized as a lexical analyzer, a parser, and an auxiliary file having functions that the parser calls to generate machine instructions, evaluate instruction and function times, and make the timing report. A listing of the documented source code for the MTE can be found in Appendix 3. There is also a parameter file that is part of the MTE source program and is described in section 2.6. The MTE's main function, in the file **yacc.src**, begins by reading the instruction and function timing data into memory. It then begins lexical analysis of the C program and continues until the identifier **main** is encountered. At this point, control is passed to the parser which then processes the body of the **main** function, generating code and building the data structures required for the timing report. At the end of the **main** function, control is passed to the function that interprets the built up data structures and prints a formatted Timing Report. Note that the code generated

is used only for calculation of delays and is not printed unless **debug_flag** in the file **parameters.h** has been set to 1.

3.2 The Lexical Analyzer

The lexical analyzer of the MTE is in the file **iex.src** which is the source specification for the UNIX utility **lex**. This utility is a lexical analyzer generator which, from the input specification, produces a finite state machine, implemented in C, to perform the task of lexical analysis and puts it in the file **lex.yy.c**. For more details of the operation of **lex**, refer to the *Lex User Manual.*

The file **lex.yy.c** contains a function **yylex()** that is called anytime a new token is needed. **yylex()** returns a distinct integer for each token and -1 upon end of file. Every reserved word of the C language is a separate token; so is every operator of the language. Integer constants, floating constants, character constants, string constants, and identifiers are the other tokens returned by **yylex()**.

Any C preprocessor lines in the C program are ignored. The C program might contain comments, some of which are intended for the MTE and others that are not. Comments meant for the MTE should begin with the string **MTE**, and are used to convey either a looprange, or a filename. The lexical analyzer ignores comments without the **MTE** tag and returns the tokens **__LOOPRANGE** or **__FILENAME** for the other comments.

The lexical analyzer uses one or both of the following mechanisms to pass additional information about tokens to the parser:

1. A global variable, **yylval**, which is defined as

   struct { float time; short where; type_type type; }

   where **type_type** is currently defined as **short**. This variable is considered by the parser to be the *value* of any token that is returned by ,**yylex()**.

2. A string table which is an array of character strings and is defined in the file **aux.c**. Strings are stored into the string table in a cyclic way so that the table never overflows.

Additional information is passed only for the following tokens:

1. Identifiers, string constants, and the token **__FILENAME**. The identifier name, or the string, or the filename is copied into the string table and its index in the table is copied into the **where** field of **yylval**.
2. Constant integers and the token **__LOOPRANGE**. The actual integer is copied into the field **time** of **yylval**.
3. Additional information is not passed for any other token, since it is not needed.

The lexical analyzer has no access to any symbol table; this is one of the reasons for the MTE not being able to handle types defined by the user through **typedef** statements. The lexical analyzer cannot distinguish between an identifier and a defined type.

3.3 The Parser

The parser of the MTE is in the file **yacc.src** which is the source specification for the UNIX utility ,**yacc**. This utility is a compiler compiler that, given a context free grammar, generates a C program to parse the language defined by the grammar. It places its output in the file **y.tab.c** that contains a function **yyparse()** which performs the task of parsing. **yacc** also has provisions for specifying actions to be performed when grammar rules are reduced. For more details about this compiler compiler, refer to the *Yacc User Manual*.

The grammar used for implementing the parser is not an exact grammar of the C language; the parser generated from it will also accept certain incorrect C programs. However, it has been chosen since the input C programs are expected to be correct and this inexact grammar makes implementation easier. Certain parts of the grammar have been deliberately made ambiguous; this causes **yacc** to generate conflict messages which can be ignored.

The generated parser is a bottom-up, shift reduce parser of the LALR class. The parser receives control when the) in the declaration **main**(...) has been seen. It calls a function, **yylex**() whenever it needs a new token. The parser has a value stack where it stores

the value of every nonterminal and terminal (token) that resides on the parsing stack. The type of the value stack is defined as

struct { float time; short where; type_type type; }

This definition has been used keeping in mind that the most frequently occurring nonterminal on the stack will be **exp** which stands for expression. The field **time** is used to store the time delay caused by the expression, **where** has information on where the expression currently resides (ie) how it is to be addressed, and **type** contains information about the type of the expression (ie) whether it is an array, or an integer, etc. It should be noted that in the cases of a few nonterminals, these fields are used to store other, unrelated, information.

Since the parser imposes restrictions on when actions can be performed, the MTE implementation is made more difficult. Furthermore, some nonterminals need much more information to be associated with them than the above defined structure allows. To save memory, the structure has been kept to this size since these nonterminals are not expected to occur frequently on the stack. Since **yyparse()** controls the order of events, recursion cannot be used and instead, explicit stacks have to be used to handle certain other nonterminals.

3.4 Symbol Tables and Types

The MTE maps all variable declarations in the input C program to two basic types, integer and floating, and two storage classes , static and automatic. It also recognizes arrays, functions, and structures of these types. The default type for any variable is integer, and the default storage class is automatic; (ie) variables declared (or mapped by MTE) as such are not stored in any symbol table and , to the MTE, are indistinguishable from variables occurring in the **main()** function that have not been declared in this function. Since the majority of variables are expected to be of the default storage class and type, this strategy prevents cluttering of the symbol tables and saves space.

The C types **char**, **short**, **unsigned**, **int**, and **long** are all considered to be of type integer and the types **float** and **double** are mapped to the MTE type floating. The C storage classes **static** and **external** are considered to be static and all oter storage classes are considered to be of the class automatic. Note that variables declared **register** are also considered automatic because the C compiler does not guarantee that these variables will be placed in registers.

There are two symbol tables, implemented by open hashing, one for storing variables whose storage class is static and one for variables whose type is not integer. The second table contains type information for each variable entered into it, type information being encoded so that the following types can be distinguished between :

1. Floating scalars.
2. Floating arrays - the number of dimensions of the array is also encoded.
3. Integer arrays - the number of dimensions is also encoded.
4. Character arrays - both the number of dimensions of the array and the size of the last dimension (for use in evaluating function timings) are encoded. This is the only case in which the C types **int** and **char** are differentiated between.
5. All structures - there is a separate array that stores structure definitions, with each element of the array being the header for a linked list of fields of non default types belonging to a single structure. It is the index into this array that is encoded as type information for variables declared as structures.

The justification for the above choice of types and storage classes will be more apparent in the next section, where the usage of the symbol tables during code generation is explained. The functions used for processing of variable declarations, in the file **aux.c**, are tab_init, hash, insert, search, new, link, getfield, getstruc, findstruc, s_proc, & update.

3.5 Code Generation

The target instruction set and addressing modes are listed in chapter 2. Briefly, the MTE's internal storage classes, static and automatic, correspond to the addressing modes ABSolute and REGister_RELative. The mode IMDTE is used for literal constants and REG is for intermediate results stored in the processor registers. The types integer and floating correspond to the two different types of instructions in the target instruction set.

There is no building up of elaborate data structures or parse trees before code is generated. The MTE tries to imitate the Sequent C compiler to the extent permitted by its limited instruction set and addressing modes. Optimization is not done to a great extent, the reason being that it is better to err on the conservative side. Depending on the types of the operands, integer or floating instructions are generated and code to perform necessary type conversions is also generated. One assumption behind the mapping of all integral types to one type, integer, is that the C type **int** occupies a word or less of memory & operations on it take the same time as operations on smaller types such as **char** and **short**.

The MTE does not keep track of register usage during code generation; it does not even know the total number of processor registers available on the target machine. Instead, it assumes that a register is available whenever an intermediate result has to be stored. This assumption is justified in most cases, since normally, expressions are not long enough for the number of intermediate results to exceed the number of available registers. Another assumption is that parameters are passed to functions by pushing them onto the stack, and functions return their values in a register.

Code generation for most statements is self-explanatory and obvious. A few points to note about code generation for expressions :

1. Identifiers - both symbol tables are searched to determine the type and addressing mode and this information is filled into the **type** and **where** fields of its value. **time** is set to 0. A similar procedure is followed for constants except that searching of symbol

tables is not necessary.

2. Function calls - code is generated to push the parameters, if any, onto the stack and then a function call instruction is generated. The type symbol table is searched to determine the type of the value returned by the function.
3. Array expression - Address calculation code is generated. Optimization is done in the sense that calculations that can be performed at compile time are assumed to be done by the compiler. MTE also keeps track of whether the expression is an address or whether it refers to an object in the array.
4. Structure expression - the named field is searched for in the array containing the structure definitions, and if present, its type is copied into the **type** for the expression. The type encoding for fields in the structure definitions array is exactly the same as the type encoding for variables in the symbol table. This is how nested structures are handled. No code is generated since the address calculation can be done at compile time.
5. Relational operators - the operands are checked to make sure that they are addressable; if not instructions are generated to make them addressable. Type conversions, if necessary, are performed and a CoMPare instruction is generated. The relational expressions's **where** field is set equal to "PSW" which stands for processor status word. This is done because the expression could either be used to set or clear a variable, or to alter the flow of control. The first case would require a conditional set instruction while the second case would need a conditional branch instruction. When an expression is in the PSW, it is not in an addressable form.
6. Unary operators - code is generated immediately rather than waiting to see what the unary expression is to be used for. An exception is when the operator is logical not and the operand is in the PSW. In this case no code is generated since the condition in the following conditional instruction just has to be reversed.

7. Binary arithmetic operators - both operands are converted to addressable forms and necessary type conversion code is generated. If the operator is commutative, at least one operand has to be in a register; if it is noncommutative, one particular operand has to be in a register. Code is generated to ensure this before the instruction to perform the actual operation is generated. The result is always stored in a register.
8. Binary logical operators - code is produced to ensure that both operands reside in the PSW. A conditional branch instruction is generated and the logical expression's **where** field is set to PSW.
9. Assignment operators - if the operator is not simple assignment (=), code generation proceeds as in the case of binary arithmetic operators except that the destination is not a register but a memory location. Expressions with the equal to operator are optimized to a greater extent since they are very frequent.

Finally, it should be noted that instructions are not always generated in the correct order as the MTE cares only about the total time delay. The routines used in code generation, in the file **aux.c**, are get_lvalue, get_ops, adjst_types, arrayint, flotint, arithop, relop, asgnop, & logic.

3.6 Timing Evaluation

When the MTE starts executing, it reads 279 numbers from the instruction timing data file into a single linear array. Function timing expressions are read from the function timing data file into an array of strings, with one line of input data, having a function name and its timing expression, being stored in one string. At this point, the syntax of the timing expressions are also checked. Then the array of strings is sorted by function name to make later searching faster.

Whenever the MTE generates an instruction, it actually calls the function **eval** which returns a floating point number that is the time in microseconds of instruction execution. **eval** takes 4 parameters - the instruction code, its type, and 2 addressing modes for its two

operands. A dummy parameter, DUM, is passed to this function in place of one or more of the last 3 parameters if they are not applicable. If the instruction is a function call, **eval** calls a different function to evaluate the delay caused by the function. Otherwise, based on the parameters it receives, eval calculates an index into the array storing the instruction times and returns the time there.

As the input program is being processed, whenever a function call is recognized, (ie) the construct *identifier(* is seen, an entry for the function is made on top of a stack of entries. An entry for a function contains the function name and space for storing the values of its arguments (as defined in chapter 2), and the number of arguments in the function call.The stack is necessitated because an argument to a function might itself be a call to another function. As the arguments in a function call are parsed, their values (if any) are evaluated and stored in the entry on top of the stack. When the end of the function call is recognized, **eval** is called to evaluate the function delay and finally, the entry on top of the stack is popped off.

When **eval** is called to evaluate function delays, it calls a function **f_proc** to do the task, with the implicit understanding that the entry for the function resides on top of the stack. **f_proc** calls other functions to do a binary search on the array of strings having the function times to retrieve the timing expression, substitute for any argument values in the expression, and evaluate the resulting arithmetic expression. Lastly, **f_proc** checks if the function name is present in the array **special_funcs** to determine if the function is a critical event. If so, it calls other functions to create a node for the event, as explained in the next section. The calculated delay is then returned to the calling function.

The functions, in file **aux.c**, that are used for timing evaluation are i_init, eval, f_init, f_cmp, valid_exp, get_index, f_proc, f_calc, bin_search, & exp_eval.

3.7 Timing Report generation

While the program is being parsed, a linked list of *nodes* is constructed, with each node corresponding to either a critical event or a control event (loops). This list is made up of nodes of 2 types, **io_type** and **control_type**, and the list begins and ends with 2 nodes of **io_type** corresponding to the program beginning and end.**io_type** nodes can store a nodenumber, arrival time, delay time, function name, filename, a stack of integers and a pointer to the next node. There is a global variable, **clock**, that is used to keep track of elapsed time in the program. Whenever the function **eval** is called to evaluate an instruction or function time delay, **clock** is incremented by the calculated delay. This variable is also adjusted elsewhere to account for conditional and looping program structures. Whenever an **io_type** node is attached to the list, it is given a unique positive integer as its nodenumber while a **control_type** node has a label that denotes what kind of control event it corresponds to.

Briefly, the linked list of nodes is constructed as follows.

1. Before parsing begins, the list contains one node of io_type that corresponds to program beginning and has a nodenumber of 0.
2. Whenever a critical event is recognized, a node is attached to the list, with the value of **clock** being its arrival time and the stack **if_stack** being copied into its stack of integers. Each "if" statement is assigned a unique integer and the if_stack has the numbers of all the enclosing "if" statements at any point in the program. The number is positive if the enclosing statement is the "then" statement and negative if the enclosing statement is the "else" part of the "if" statement. Each critical event has a copy of this stack at the point it is encountered in the program so that mutually exclusive critical events can be recognized as such by comparing their stacks. **switch** statements are treated in exactly the same way as nested "if-then-else" statements.
3. Whenever a loop statement is encountered either the combination of

WHLBEGIN-WHLEXIT-WHLEND or DOBEGIN-DOEND **control_type** nodes is created depending on whether the loop is a **while** loop or a **do-while** loop.**for** loops are treated the same way as "while" loops as implied by the definition of "for" loops. The nodes WHLBEGIN or DOBEGIN are created before the loop is parsed, and the nodes WHLEND and DOEND are created after the body of the loop has been parsed. The WHLEXIT node is created after the loop control expression has been parsed, but before the body of the loop is parsed. In all cases, the "arrival" time is the value of the clock when the node is created, and the field "exit", used only for the WHLEXIT and DOEND nodes, is the time the loop is exited.

4. **clock** is always set to 0 before a loop statement is parsed. Hence, all nodes within the loop statement will have their arrival time marked relative to the time of arrival at the top of the loop. The "clock" variable is set to the loop exit time at the end of the loop.
5. At the end of the program, the node corresponding to program end is attached to the list.

Finally, using only the information in this linked list, a complicated recursive function calculates time delays between all pairs of critical events according to the rules listed in section 2.5. The functions used for timing report generation are makenode, wind_up, access, t_calc, node_calc, line, & print_report.

CHAPTER 4

LIMITATIONS OF THE MTE

4.1 Introduction

This chapter describes the shortcomings of the MTE in its current implementation and suggests modifications that might improve its performance. The task of estimating a program's execution time without actually executing it is inherently difficult because most modern day computers have complex operating environments whose effect on the program execution time is not easily predicted. The estimation is made even more difficult (and inaccurate) because of the requirement that the MTE be portable (ie) be able to predict execution times for different target machines.

The decision to have the input to the MTE as the C programs produced by the MODEL compiler, and not the user provided MODEL specification, or the MODEL compiler generated intermediate level flowchart, or the machine language program produced by the C compiler, was taken due to a number of reasons. The higher the level of input, the greater the duplication of work will be and the greater the inaccuracy will be. On the other hand, a very low level input such as machine language program results in a loss of information about program structure and a loss of portability. Hence, a compromise between accuracy and portability was reached by starting from the C program as input.

4.2 The Target Machine

The following is a list of assumptions about the target machine's environment that may cause inaccuracies.

1. The machine is dedicated to the program it is executing, (ie) whenever the program wants CPU time, it obtains it. This is not true of multiprogrammed systems; however, large real time applications do usually run on dedicated systems.
2. The entire program resides in main memory all the time; delays that may be caused by

page faults during program execution are not accounted for.

3. The program does not have to wait in any queue for input/ output operations. All the resources of the computer should be dedicated to the program.
4. If the program is executing on a multi-processor machine and communicates with other programs running on other processors, delays caused by waiting for the other programs to be ready to communicate are not accounted for.
5. The instruction set and addressing modes that most machines have are much more rich than the assumed target instruction set. For example, the Sequent has some "quick" instructions for small integers and 2 additional addressing modes which could make programs execute faster.

4.3 MTE Internal Inaccuracies

In order to ease the task of designing and implementing the MTE, a few assumptions have been made that may cause inaccuracies :

1. The MTE maps the C type **double** to floating instructions. This is because frequent use of the above data type is not expected; however, if the user wants a very conservative estimate, he would just have to alter the time for every floating instruction in the instruction timing data file by substituting the time for the corresponding "double" instruction.
2. The MTE expects a constant integer to be specified as the range for each loop. The number of iterations for each loop may not always be known - in this case, a maximum number is specified which may not reflect the true number of iterations.
3. Complex optimizations are not performed on the code generated. Realistically, the C programs are likely to be compiled with the optimizing option to obtain the final executable version. So, the generated code will be close to the unoptimized version and not the final version.

4.4 Future Modifications

Here are some modifications that could be made to the MTE to improve its accuracy :

1. The types of the target instruction set could be enriched without much danger of losing portability.
2. It might be possible to split individual instruction times into a fetching and executing time, a fetching time for the source operand, and a fetch and store time for the destination operand. This way, the size of the instruction timing data file could be considerably reduced from its present cumbersome size of 279 numbers.
3. The syntax of function timing expressions in the function timing data file could be made more powerful. At present, it is difficult to accurately express timings for functions like **printf** and **scanf** with the current syntax.
4. Code generated could be optimized more to reflect the true code generated by the actual C compiler, (ie) the MTE should be tuned according to the nature of the C compiler that generates code for the target machine in question.
5. Each looprange could be considered a variable and time delays could be expressed as functions of these variables. The user could then manually substitute the actual loopranges for each execution of the program and arrive at a better estimate for the time of that particular program execution.

APPENDIX 1

Data files

1. Instruction Timings :

```
/****************** FILE "idata.template" **********************
 Nov 17th, '87                                        Mahesh Srinivasan

     This is a template file for instruction timing data for
the MODEL Timing Evaluator. To obtain the actual timing
data file, idata, run the C preprocessor on this file to
remove the comments, redirecting the output into the
file idata.IMPORTANT : After doing the above, the first
line in the file idata should be deleted because it is a
"#" line, placed there by the C preprocessor.
     The times given below are for the Sequent Balance 21000
machine. For a different target machine, only the numbers
outside the comments need to be changed. This file should
always contain 279 numbers, outside comments, which are
interpreted by the MTE as the execution times, in MICROSECONDS,
of the instructions along which the numbers appear. For more
information about the instruction set and addressing modes,
refer to the report, "MODEL TIMING EVALUATOR"
    Instructions ending with an "i" have operands equal to the
size of the C data type "int" on the target machine. Those that
end with "f" correspond to the C data type "float" and the
instructions with neither of these endings are those for which
no type is applicable, or only one type is possible and that is
"int"
    Addressing modes
1   IMDTE    a constant
2   ABS      the address of the operand
3   REG      the operand resides in a register
4   REG REL    register relative. The address of the operand is
                formed by adding a constant offset to the contents
                of a register

            ALL TIMES IN MICROSECONDS
*************************************************************/

/* The following 2 lines are not really machine instructions, but
correspond to the time delay caused by the operating system in
beginning and ending the program execution. If the times are not
significant, or unimportant, they can be set to 0                  */
/*PBEGIN          */                  0
/*PEND            */                  0

/* RET is the return from function instruction. For lack of a
   better estimate, being set to time of unconditional branch    */
/*RET              */                  1.915

/* Unconditional branch, unsuccessful conditional branch, and
successful conditional branch
/*BR                          */              1.915
```

```
/*BRcond FAIL            */          0.723
/*BRcond SUCC            */          2.174

/* Scond, after a compare instruction, sets or clears its
operand, depending on whether "cond" is true or false          */
/*Scond REG             */          1.239
/*Scond ABS             */          2.115
/*Scond REG_REL         */          1.989

/* ADDR places the address of its source operand in its
destination operand.  Note that the source operand cannot be
IMDTE or REG                                                   */
/*ADDR   ABS      ,REG      */          1.083
/*ADDR   ABS      ,ABS      */          2.22
/*ADDR   ABS      ,REG_REL  */          2.066
/*ADDR   REG_REL,REG        */          0.827
/*ADDR   REG_REL,ABS        */          2.063
/*ADDR   REG_REL,REG_REL    */          1.753

/* MODulus    dest = dest % src                                */
/*MOD      REG      ,REG      */          14.457
/*MOD      REG      ,ABS      */          14.876
/*MOD      REG      ,REG_REL  */          14.666
/*MOD      ABS      ,REG      */          13.791
/*MOD      ABS      ,ABS      */          14.051
/*MOD      ABS      ,REG_REL  */          13.531
/*MOD      REG_REL,REG        */          13.531
/*MOD      REG_REL,ABS        */          13.737
/*MOD      REG_REL,REG_REL    */          13.322
/*MOD      IMDTE    ,REG      */          13.222
/*MOD      IMDTE    ,ABS      */          13.919
/*MOD      IMDTE    ,REG_REL  */          13.43

/* Bitwise AND    dest = dest & src                             */
/*AND      REG      ,REG      */          0.413
/*AND      REG      ,ABS      */          2.064
/*AND      REG      ,REG_REL  */          1.857
/*AND      ABS      ,REG      */          1.653
/*AND      ABS      ,ABS      */          3.091
/*AND      ABS      ,REG_REL  */          2.736
/*AND      REG_REL,REG        */          1.522
/*AND      REG_REL,ABS        */          2.729
/*AND      REG_REL,REG_REL    */          2.218
/*AND      IMDTE    ,REG      */          1.135
/*AND      IMDTE    ,ABS      */          2.578
/*AND      IMDTE    ,REG_REL  */          2.347

/* Bitwise OR    dest = dest | src                              */
/*OR       REG      ,REG      */          0.412
/*OR       REG      ,ABS      */          2.064
/*OR       REG      ,REG_REL  */          1.857
```

```
/*OR      ABS      ,REG      */          1.651
/*OR      ABS      ,ABS      */          3.091
/*OR      ABS      ,REG_REL  */          2.735
/*OR      REG_REL,REG       */         1.522
/*OR      REG_REL,ABS       */         2.726
/*OR      REG_REL,REG_REL   */         2.218
/*OR      IMDTE    ,REG      */          1.135
/*OR      IMDTE    ,ABS      */          2.578
/*OR      IMDTE    ,REG_REL  */          2.345

/* Bitwise eXclusive OR : dest = dest ^ src                              */
/*XOR     REG      ,REG      */          0.412
/*XOR     REG      ,ABS      */          2.065
/*XOR     REG      ,REG_REL  */          1.856
/*XOR     ABS      ,REG      */          1.651
/*XOR     ABS      ,ABS      */          3.093
/*XOR     ABS      ,REG_REL  */          2.735
/*XOR     REG_REL,REG       */         1.522
/*XOR     REG_REL,ABS       */         2.727
/*XOR     REG_REL,REG_REL   */         2.219
/*XOR     IMDTE    ,REG      */          1.134
/*XOR     IMDTE    ,ABS      */          2.578
/*XOR     IMDTE    ,REG_REL  */          2.347

/* One's COMPlement : dest = ~ src                                       */
/*COMP    REG      ,REG      */          1.345
/*COMP    REG      ,ABS      */          2.173
/*COMP    REG      ,REG_REL  */          2.378
/*COMP    ABS      ,REG      */          2.096
/*COMP    ABS      ,ABS      */          3.049
/*COMP    ABS      ,REG_REL  */          2.792
/*COMP    REG_REL,REG       */         1.758
/*COMP    REG_REL,ABS       */         2.272
/*COMP    REG_REL,REG_REL   */         2.481
/*COMP    IMDTE    ,REG      */          1.732
/*COMP    IMDTE    ,ABS      */          2.845
/*COMP    IMDTE    ,REG_REL  */          2.69

/* Arithmetic SHift : dest = dest << src (or dest >> src).
  Assume the shift is by the maximum number possible for "int" */
/*ASH     REG      ,REG      */          1.447
/*ASH     REG      ,ABS      */          2.145
/*ASH     REG      ,REG_REL  */          2.066
/*ASH     ABS      ,REG      */          2.95
/*ASH     ABS      ,ABS      */          3.431
/*ASH     ABS      ,REG_REL  */          3.507
/*ASH     REG_REL,REG       */         2.687
/*ASH     REG_REL,ABS       */         2.791
/*ASH     REG_REL,REG_REL   */         2.505
/*ASH     IMDTE    ,REG      */          2.169
/*ASH     IMDTE    ,ABS      */          2.582
```

```
/*ASH      IMDTE   ,REG_REL */          2.428

/* Integer TO Floating : dest = (float) src                    */
/*ITOF    REG      ,REG      */          7.38
/*ITOF    REG      ,ABS      */          9.11
/*ITOF    REG      ,REG_REL  */          8.775
/*ITOF    ABS      ,REG      */          8.621
/*ITOF    ABS      ,ABS      */          10.092
/*ITOF    ABS      ,REG_REL  */          10.123
/*ITOF    REG_REL,REG       */         8.26
/*ITOF    REG_REL,ABS       */         9.499
/*ITOF    REG_REL,REG_REL   */         9.498
/*ITOF    IMDTE   ,REG      */          8.002
/*ITOF    IMDTE   ,ABS      */          10.02
/*ITOF    IMDTE   ,REG_REL  */          9.706

/* Floating TO Integer : dest = (int) src                      */
/*FTOI    REG      ,REG      */          7.099
/*FTOI    REG      ,ABS      */          7.525
/*FTOI    REG      ,REG_REL  */          7.437
/*FTOI    ABS      ,REG      */          8.958
/*FTOI    ABS      ,ABS      */          9.269
/*FTOI    ABS      ,REG_REL  */          9.707
/*FTOI    REG_REL,REG       */         8.362
/*FTOI    REG_REL,ABS       */         8.882
/*FTOI    REG_REL,REG_REL   */         9.011
/*FTOI    IMDTE   ,REG      */          8.414
/*FTOI    IMDTE   ,ABS      */          8.743
/*FTOI    IMDTE   ,REG_REL  */          9.088

/* MOVe : dest = src                                           */
/*MOVi    REG      ,REG      */          0.387
/*MOVi    REG      ,ABS      */          1.857
/*MOVi    REG      ,REG_REL  */          1.598
/*MOVi    ABS      ,REG      */          1.601
/*MOVi    ABS      ,ABS      */          2.424
/*MOVi    ABS      ,REG_REL  */          2.372
/*MOVi    REG_REL,REG       */         1.418
/*MOVi    REG_REL,ABS       */         2.217
/*MOVi    REG_REL,REG_REL   */         1.754
/*MOVi    IMDTE   ,REG      */          1.135
/*MOVi    IMDTE   ,ABS      */          2.273
/*MOVi    IMDTE   ,REG_REL  */          2.04
/*MOVf    REG      ,REG      */          2.682
/*MOVf    REG      ,ABS      */          5.006
/*MOVf    REG      ,REG_REL  */          4.646
/*MOVf    ABS      ,REG      */          4.62
/*MOVf    ABS      ,ABS      */          6.374
/*MOVf    ABS      ,REG_REL  */          6.611
/*MOVf    REG_REL,REG       */         4.025
/*MOVf    REG_REL,ABS       */         5.883
```

```
/*MOVf   REG_REL,REG_REL  */          6.012
/*MOVf   IMDTE   ,REG     */           3.975
/*MOVf   IMDTE   ,ABS     */           5.834
/*MOVf   IMDTE   ,REG_REL */           6.094

/* ADDition : dest = dest + src                                        */
/*ADDi    REG     ,REG     */           0.412
/*ADDi    REG     ,ABS     */           2.065
/*ADDi    REG     ,REG_REL */           1.857
/*ADDi    ABS     ,REG     */           1.651
/*ADDi    ABS     ,ABS     */           3.09
/*ADDi    ABS     ,REG_REL */           2.737
/*ADDi    REG_REL,REG      */          1.522
/*ADDi    REG_REL,ABS      */          2.727
/*ADDi    REG_REL,REG_REL  */          2.219
/*ADDi    IMDTE   ,REG     */           1.136
/*ADDi    IMDTE   ,ABS     */           2.578
/*ADDi    IMDTE   ,REG_REL */           2.345
/*ADDf    REG     ,REG     */           6.515
/*ADDf    REG     ,ABS     */           10 517
/*ADDf    REG     ,REG_REL */           10.054
/*ADDf    ABS     ,REG     */           8.424
/*ADDf    ABS     ,ABS     */           11 73
/*ADDf    ABS     ,REG_REL */           11 607
/*ADDf    REG_REL,REG      */          7.884
/*ADDf    REG_REL,ABS      */          11 394
/*ADDf    REG_REL,REG_REL  */          11 022
/*ADDf    IMDTE   ,REG     */           7 836
/*ADDf    IMDTE   ,ABS     */           11 345
/*ADDf    IMDTE   ,REG_REL */           11 091

/* SUBtraction    dest = dest    src                                  */
/*SUBi    REG     ,REG     */           0 412
/*SUBi    REG     ,ABS     */           2 064
/*SUBi    REG     ,REG_REL */           1 856
/*SUBi    ABS     ,REG     */           1 651
/*SUBi    ABS     ,ABS     */           3 092
/*SUBi    ABS     ,REG_REL */           2 736
/*SUBi    REG_REL,REG      */          1 522
/*SUBi    REG_REL,ABS      */          2 727
/*SUBi    REG_REL,REG_REL  */          2 219
/*SUBi    IMDTE   ,REG     */           1 135
/*SUBi    IMDTE   ,ABS     */           2 578
/*SUBi    IMDTE   ,REG_REL */           2 345
/*SUBf    REG     ,REG     */           6 515
/*SUBf    REG     ,ABS     */           10 517
/*SUBf    REG     ,REG_REL */           10 052
/*SUBf    ABS     ,REG     */           8 478
/*SUBf    ABS     ,ABS     */           11 73
/*SUBf    ABS     ,REG_REL */           11 605
/*SUBf    REG_REL,REG      */          7 987
```

```
/*SUBf   REG_REL,ABS      */          11 291
/*SUBf   REG_REL,REG_REL  */          10 981
/*SUBf   IMDTE   ,REG     */           7 832
/*SUBf   IMDTE   ,ABS     */          11 242
/*SUBf   IMDTE   ,REG_REL */          11 085

/* MULtiplication : dest = dest * src                              */
/*MULi   REG     ,REG     */           8 881
/*MULi   REG     ,ABS     */           9 195
/*MULi   REG     ,REG_REL */           9 098
/*MULi   ABS     ,REG     */           9 796
/*MULi   ABS     ,ABS     */           9 989
/*MULi   ABS     ,REG_REL */           9 51
/*MULi   REG_REL,REG      */          9 505
/*MULi   REG_REL,ABS      */          9 402
/*MULi   REG_REL,REG_REL  */          9 299
/*MULi   IMDTE   ,REG     */           9 211
/*MULi   IMDTE   ,ABS     */           9 849
/*MULi   IMDTE   ,REG_REL */           9 406
/*MULf   REG     ,REG     */           5 516
/*MULf   REG     ,ABS     */           9 47
/*MULf   REG     ,REG_REL */           9 076
/*MULf   ABS     ,REG     */           7 404
/*MULf   ABS     ,ABS     */           10 662
/*MULf   ABS     ,REG_REL */           10 576
/*MULf   REG_REL,REG      */          6 84
/*MULf   REG_REL,ABS      */          10 323
/*MULf   REG_REL,REG_REL  */          9 925
/*MULf   IMDTE   ,REG     */           6 761
/*MULf   IMDTE   ,ABS     */           10 171
/*MULf   IMDTE   ,REG_REL */           10 123

/* DIVision    dest = dest / src                                   */
/*DIVi   REG     ,REG     */          13 741
/*DIVi   REG     ,ABS     */          14 167
/*DIVi   REG     ,REG_REL */          13 952
/*DIVi   ABS     ,REG     */          14 776
/*DIVi   ABS     ,ABS     */          15 011
/*DIVi   ABS     ,REG_REL */          14 521
/*DIVi   REG_REL,REG      */          14 535
/*DIVi   REG_REL,ABS      */          14 757
/*DIVi   REG_REL,REG_REL  */          14 352
/*DIVi   IMDTE   ,REG     */          14 215
/*DIVi   IMDTE   ,ABS     */          14 884
/*DIVi   IMDTE   ,REG_REL */          14 404
/*DIVf   REG     ,REG     */           8 156
/*DIVf   REG     ,ABS     */          12 239
/*DIVf   REG     ,REG_REL */          11 666
/*DIVf   ABS     ,REG     */           9 988
/*DIVf   ABS     ,ABS     */          13 356
/*DIVf   ABS     ,REG_REL */          13 218
```

```
/*DIVf   REG_REL,REG      */          9.601
/*DIVf   REG_REL,ABS      */          12.907
/*DIVf   REG_REL,REG_REL  */          12.699
/*DIVf   IMDTE   ,REG     */          9.448
/*DIVf   IMDTE   ,ABS     */          12.856
/*DIVf   IMDTE   ,REG_REL */          12.804

/* NEGation : dest = - src                                          */
/*NEGi   REG     ,REG     */          1.138
/*NEGi   REG     ,ABS     */          1.988
/*NEGi   REG     ,REG_REL */          1.757
/*NEGi   ABS     ,REG     */          1.963
/*NEGi   ABS     ,ABS     */          2.945
/*NEGi   ABS     ,REG_REL */          2.584
/*NEGi   REG_REL,REG      */          1.55
/*NEGi   REG_REL,ABS      */          2.067
/*NEGi   REG_REL,REG_REL  */          2.093
/*NEGi   IMDTE   ,REG     */          1.628
/*NEGi   IMDTE   ,ABS     */          2.686
/*NEGi   IMDTE   ,REG_REL */          2.376
/*NEGf   REG     ,REG     */          2.863
/*NEGf   REG     ,ABS     */          5.008
/*NEGf   REG     ,REG_REL */          4.643
/*NEGf   ABS     ,REG     */          4.722
/*NEGf   ABS     ,ABS     */          6.504
/*NEGf   ABS     ,REG_REL */          6.921
/*NEGf   REG_REL,REG      */          4.339
/*NEGf   REG_REL,ABS      */          6.091
/*NEGf   REG_REL,REG_REL  */          6.116
/*NEGf   IMDTE   ,REG     */          4.181
/*NEGf   IMDTE   ,ABS     */          6.039
/*NEGf   IMDTE   ,REG_REL */          6.296

/* CoMPare instruction : sets PSW bits according to which
  operand is greater in magnitude.                                  */
/*CMPi   REG     ,REG     */          0.387
/*CMPi   REG     ,ABS     */          1.807
/*CMPi   REG     ,REG_REL */          1.677
/*CMPi   ABS     ,REG     */          1.599
/*CMPi   ABS     ,ABS     */          2.531
/*CMPi   ABS     ,REG_REL */          2.377
/*CMPi   REG_REL,REG      */          1.419
/*CMPi   REG_REL,ABS      */          2.168
/*CMPi   REG_REL,REG_REL  */          1.96
/*CMPi   IMDTE   ,REG     */          1.135
/*CMPi   IMDTE   ,ABS     */          2.325
/*CMPi   IMDTE   ,REG_REL */          2.193
/*CMPf   REG     ,REG     */          5.032
/*CMPf   REG     ,ABS     */          7.279
/*CMPf   REG     ,REG_REL */          6.723
/*CMPf   ABS     ,REG     */          6.889
```

```
/*CMPf   ABS      ,ABS      */          8.387
/*CMPf   ABS      ,REG_REL  */          8.26
/*CMPf   REG_REL,REG      */          6.4
/*CMPf   REG_REL,ABS      */          7.949
/*CMPf   REG_REL,REG_REL  */          7.743
/*CMPf   IMDTE    ,REG      */          6.248
/*CMPf   IMDTE    ,ABS      */          7.901
/*CMPf   IMDTE    ,REG_REL  */          7.849

/*                              ******************                    */
```

2. Function timings :

```
/****************** FILE "fdata.template" ********************
 Nov 20th, '87                                    Mahesh Srinivasan

      This is a template file for function timing data for
the MODEL Timing Evaluator. To obtain the actual data file,
fdata, run the C preprocessor on this file to remove the
comments, redirecting the output to file fdata.
IMPORTANT : After doing the above, the first line in the
            file fdata should be deleted since it is a
            "#" line, placed there by the C preprocessor.

      The timing expressions given below are for the Sequent
Balance 21000 machine. This file is not yet complete because
the entire list of functions that will be used by the MODEL
compiler is not available at this time. The following is a
subset of C library functions that are likely to be used.
The times for user defined functions will also have to be
explicitly entered in this file. Each line of the file
that is not part of a comment should either be blank, or
have a function name, followed by its timing expression.
For details of the syntax and semantics of the expressions,
refer to the report, "MODEL TIMING EVALUATOR".

                    ALL TIMES IN MICROSECONDS
All constants should be in decimal format!!!
*****************************************************************/

/*              STRING functions                        */
strcpy          24.5 + 3.4*a2
strncpy          29.3 + 3.4*a3
strcat          30.5 + 3.5*a1 - 0.1*a2
strncat          33.5 + 3.5*a1 - 0.1*a3
strcmp          26.8 + 4.8*a1
strncmp          29 + 4.8*a3
strlen          22 + 3.5*a1

/*            MATH functions              */
sin         409.4
cos         417.6
tan         429
exp         250.3
log         263.5
log10         273
sqrt        170.4
pow         436.9
fabs         11.3
floor         164.1
ceil        233.7
```

```
/*       File I/O functions     */
creat      8498.5
open      4772.4
close      294.9
read      2217 + 5.2*a3
write      2312 + 5.3*a3

/*                          **********                  */
```

APPENDIX 2

Example program

1. **MODEL Specification :**

```
MODULE: TEST;
SOURCE: ACMIN;
TARGET: ACMOUT;
ACMIN is FILE   RECORD INREC;
INREC IS RECORD (M,N);
    M IS FIELD (NUMERIC(4));
    N IS FIELD (NUMERIC(4));
ACMOUT is FILE   RECORD OUTREC;

OUTREC IS RECORD (ANSWER);
    ANSWER IS FIELD (NUMERIC(8));
STACKS IS GROUP (STACK(*));
    STACK   IS GROUP (TOP,STACKE(1:200));
            STACKE IS FIELD (NUMERIC(8));
            TOP     IS FIELD (NUMERIC(8));
/*********************************************************/
S1 IS SUBSCRIPT(*);
S2 IS SUBSCRIPT(*);
IF TOP(S1)=1 THEN ANSWER=STACKE(S1,1);
/* DEFINE SIZE OF STACKE */
TOP(S1)=IF  S1=1 THEN 2
            ELSE IF STACKE(S1-1,TOP(S1-1)-1)=0 THEN TOP(S1-1)-1
            ELSE IF STACKE(S1-1,TOP(S1-1))=0 THEN TOP(S1-1)
            ELSE TOP(S1-1)+1;
/* DEFINE STACK CONTENT */
STACKE(S1,S2)=IF  S1=1 THEN IF S2=1 THEN ACMIN.M ELSE ACMIN.N
                    ELSE IF STACKE(S1-1,TOP(S1-1)-1)=0 THEN
                           IF S2=TOP(S1) THEN STACKE(S1-1,S2+1)+1
                                         ELSE STACKE(S1-1,S2)
                    ELSE IF STACKE(S1-1,TOP(S1-1))=0 THEN
                           IF S2=TOP(S1) THEN 1
                           ELSE IF S2=TOP(S1)-1 THEN STACKE(S1-1,S2)-1
                           ELSE STACKE(S1-1,S2)
                    ELSE IF S2=TOP(S1) THEN STACKE(S1-1,S2-1)-1
                           ELSE IF S2=TOP(S1)-1 THEN STACKE(S1-1,S2-1)
                           ELSE IF S2=TOP(S1)-2 THEN STACKE(S1-1,S2)-1
                           ELSE STACKE(S1-1,S2);
/* DEFINE RANGE OF STACK */
SIZE STACKE(S1)= TOP(S1);
END STACK(S1)=      TOP(S1)=1;
```

2. The C program :

```
int errpic = 0;
int _si;
char *_ptr;
#include <stdio.h>
 main(argc,argv) /* program TEST */
  int argc;
  char argv[];
  {
     int ACMINS;
     short _FSTACMINS = 1;
     short ENDFILE_ACMINS = 0;
     char _FB36[9];
     int _FX36;
     char _RV37[9];
     int _RX37;
     int _su,_sb;
     int _EOF49;
     int _W_D49[3];
     char _RV41[10];
     int _RX41;
     int ACMOUTT;
     short _FSTACMOUTT = 1;
     FILE *fperr, *fopen();
     static short ERRORF_BIT = 0;
     char _ERROR_BUF[270];
     short _NOT_DONE[20];
     char _038_M [005];
     char _039_N [005];
     char _042_ANSWER [010];
     long _048_TOP[3];
     long _049_STACKE[3][201];
     short  _050_END;
     long  _051_SIZE;
     long _I1;
     long _I2;
     int _enci();
     double _encr();
     char *_dec();
     int _INITWIN();

        _FB36[1]='\0';
        _INITWIN(_W_D49,2);
        ACMOUTT=creat("ACMOUTT",0777);
        if (ACMOUTT==-1) {printf("unable to open file ACMOUTT");
            exit(1);
        }
        if (argc > 1) { /* ERRORF name provided */
           ERRORF_BIT = '0';
           fperr=fopen(*argv,"w");
```

```
      }
    ACMINS=open("ACMINS",0);
    if (ACMINS==-1) {printf("unable to open file ACMINS");
        exit(1);
    }
    _sb= /*MTE ACMINS */ read(ACMINS,_RV37,8);
    if (!_sb) {ENDFILE_ACMINS = 1;
/*MTE 8*/
       for(_su=0;_su<8;_su++) _RV37[_su]=' ';
       _RV37[8]='\0';
    }
    else if (_sb<0) { printf("error in reading file ACMINS");
        exit(3);
    }
    _RX37=0;
/*MTE 8*/
    for(_su=0;_su<=8;_su++) _ERROR_BUF[_su]=_RV37[_su];
/*MTE 8*/
    for(_su=_sb;_su<8;_su++) _RV37[_su]=' ';
    _RV37[8]='\0';
/*MTE 4*/
    for (_su=0;_su<004;_su++) _038_M[_su]=_RV37[_su+_RX37];
    if (errpic) { errpic=0;
       _ptr= _dec(0.0, 0, 2, 4, 4);
/*MTE 40*/
       for (_si=0;_038_M[_si++] = *_ptr++;);
    }
    _RX37 += 004;
/*MTE 4*/
    for (_su=0;_su<004;_su++) _039_N[_su]=_RV37[_su+_RX37];
    if (errpic) { errpic=0;
       _ptr= _dec(0.0, 0, 2, 4, 4);
/*MTE 40*/
       for (_si=0;_039_N[_si++] = *_ptr++;);
    }
    _RX37 += 004;
    _I1 =0;
    _NOT_DONE[1]=1;
/*MTE 999*/
     do {_I1++;
      /* 18 */if (_I1==1) _048_TOP[2] = 2;
       else if
(_049_STACKE[_W_D49[1]][_048_TOP[1]-1]==0)
 _048_TOP[2]
      = _048_TOP[1]-1; else if
(_049_STACKE[_W_D49[1]][_048_TOP[1]]==0)
        _048_TOP[2] = _048_TOP[1]; else _048_TOP[2] =
 _048_TOP[1]+1;
      /* 20 */_051_SIZE = _048_TOP[2];
      _EOF49=_I2;
      _I2 =_048_TOP[2];
```

```
          if (!
(_I1==1))_049_STACKE[_W_D49[2]][_048_TOP[1]-1]
 =
       _049_STACKE[_W_D49[1]][_048_TOP[1]-1] ;
 if (!(_I1==1) &&

 _049_STACKE[_W_D49[2]][_048_TOP[1]-1]==0)_049_
STACKE[_W_D49[2]]
       [_I2+1] = _049_STACKE[_W_D49[1]][_I2+1];
 _I2 =_048_TOP[2];   if (
       !(_I1==1) && !(
 _049_STACKE[_W_D49[2]][_048_TOP[1]-1]==0))
       _049_STACKE[_W_D49[2]][_048_TOP[1]] =
 _049_STACKE[_W_D49[1]]
       [_048_TOP[1]]; _I2 =_048_TOP[2]-1;   if (!(_I1==1)
 && !(
       _049_STACKE[_W_D49[2]][_048_TOP[1]-1]==0)
 &&
       _049_STACKE[_W_D49[2]][_048_TOP[1]]==0 &&
 !(  _I2==_048_TOP[2]))
       _049_STACKE[_W_D49[2]][_I2] =
 049_STACKE[_W_D49[1]][_I2]; _I2
       =_048_TOP[2];   if (!(_I1==1) && !(
 049_STACKE[_W_D49[2]]
       [_048_TOP[1]-1]==0) && !(
 049_STACKE[_W_D49[2]][_048_TOP[1]]==0)
       )_049_STACKE[_W_D49[2]][_I2-1] =
 049_STACKE[_W_D49[1]][_I2-1];
       _I2 =_048_TOP[2]-1;   if (!(_I1==1) && !(
 049_STACKE[_W_D49[2]]
       [_048_TOP[1]-1]==0) && !(
 _049_STACKE[_W_D49[2]][_048_TOP[1]]==0)
        && !(
 I2== 048_TOP[2])) 049_STACKE[_W_D49[2]][_I2-1] =
       049_STACKE[_W_D49[1]][_I2-1]; _I2
  048_TOP[2]-2;   if (!(_I1==1)
        && !(
 049_STACKE[_W_D49[2]][_048_TOP[1]-1]==0) && !
(
       _049_STACKE[_W_D49[2]][_048_TOP[1]]==0) &&
 !(  _I2== 048_TOP[2])
       && !(
 I2== 048_TOP[2]-1))_049_STACKE[_W_D49[2]][_I2] =
       _049_STACKE[_W_D49[1]][_I2];
        I2 =1;
          if (_I1==1) 049_STACKE[_W_D49[1]][_I2] =  enc
( 038_M, 0, 2, 4,
        4);
/*MTE 2*/
        for (_I2 =1; _I2<= 051_SIZE, _I2++) {   if (_I1==1 && !(_I2==1))
       049_STACKE[_W_D49[1]][_I2] =  enc( 039_N, 0, 2,
```

```
4, 4),
      LOOP END2 ,)   I2 - O48 TOP[2],   if (!( I1==1) &&

  O49 STACKE[ W D49[2]][ O48 TOP[1] 1]==0) O49
STACKE[ W D49[1]]
      [ I2] =  O49 STACKE[ W D49[2]][ I2+1]+1,
  I2 = O48 TOP[2],   if (
      !( I1==1) && !(
  O49 STACKE[ W D49[2]][ O48 TOP[1] 1]==0) &&

  O49 STACKE[ W D49[2]][ O48 TOP[1]]==0) O49 ST
ACKE[ W D49[1]]
      [ I2] = 1, I2 = O48 TOP[2] 1;   if (!( I1==1) && !(
       O49 STACKE[ W D49[2]][ O48 TOP[1] 1]==0)
 &&
       O49 STACKE[ W D49[2]][ O48 TOP[1]]==0 &&
 !( I2== O48 TOP[2]))
       O49 STACKE[ W D49[1]][ I2] =
  O49 STACKE[ W D49[2]][ I2] 1;
       I2 = O48 TOP[2],   if (!( I1==1) && !(
  O49 STACKE[ W D49[2]]
      [ O48 TOP[1] 1]==0) && !(
  O49 STACKE[ W D49[2]][ O48 TOP[1]]==0)
      ) O49 STACKE[ W D49[1]][ I2] =
  O49 STACKE[ W D49[2]][ I2 1] 1;
       I2 = O48 TOP[2] 1,   if (!( I1==1) && !(
  O49 STACKE[ W D49[2]]
      [ O48 TOP[1] 1]==0) && !(
  O49 STACKE[ W D49[2]][ O48 TOP[1]]==0)
       && !(
 I2== O48 TOP[2]))O49 STACKE[ W D49[1]][ I2]
 =
       O49 STACKE[ W D49[2]][ I2 1]; I2
 = O48 TOP[2] 2;   if (!( I1==1)
       && !(
  O49 STACKE[ W D49[2]][ O48 TOP[1] 1]==0) && !
(
       O49 STACKE[ W D49[2]][ O48 TOP[1]]==0) &&
 !(  I2== O48 TOP[2])
      && !(
  I2== O48 TOP[2] 1))O49 STACKE[ W D49[1]][ I2] =
      O49 STACKE[ W D49[2]][ I2] 1;
        /* 17 */if (_O48 TOP[2]==1) {_ptr =
 O49 STACKE[ W D49[1]][1];
/*MTE 201*/
            for(_s1=0;_O42 ANSWER[_s1++]= *_ptr++;);
         }
         /* 21 */_O50 END = _O48 TOP[2]==1;
         if (_O50 END) _NOT DONE[1]=0;
         _O48 TOP[1] = _O48 TOP[2];
       }
```

```
        while (_NOT_DONE[1]);
        _RX41=0;
/*MIE 9 */
        for
(_su=0;_su<009;_su++)_RV41[_su+_RX41]=_042_ANSWER[_su];
        _RX41 += 009;
        _sb= /*MIE ACMOUTT */write(ACMOUTT,_RV41,9);
        if (_sb<9) { printf("error in writing file ACMOUTT");
            exit(4);
        }
        ACMOUTT=close("ACMOUTT");
    } /* TEST */
 int _INITWIN(WIN_VEC,LEN) /* $START$ */
 /*$PARMS: 01,9,9 */
 int *WIN_VEC, LEN;
 {
    int I;
    for (I=1; I<=LEN; I++) *(WIN_VEC+I)=I;
 }    /* $END$ */
 int _enci(ins,ldz,sgnt,dcp,ttl) /* $START$ */
 /*$PARMS: 05,0,1,1,1,1,0 */
 char ins[];
 int ldz,sgnt,dcp,ttl;
 { int val;
    double _encr();
    val=_encr(ins,ldz,sgnt,dcp,ttl);
    return(val);
 }    /* $END$ */
 double _encr(ins,ldz,sgnt,dcp,ttl) /* $START$ */
 /*$PARMS: 05,0,1,1,1,1,0 */
 char ins[];
  /* ins contains string to be translated */
                                /*       either by spaces (ldz>0) or * (ldz<0)
    */
                                /* sgnt=0 no sign, -1 only -, +1 only +, 2
 always */
                                /* if sgnt=-2 then it is a string
    */
                                /* dcp decimal point position, dec. point
 present */
                                /*       if dcp<0
    */
                                /* ttl total number of digit in the result
    */
 {
     char ch=' ';
     int i = 0,dig,sig = 1;
     double val = 0.0;
     if (sgnt = -2)
     {   sgnt = 0;
         while (ins[i] !='\0')
```

```
        { if (ins[i] == '+' || ins[i] == ' ') sgnt=2;
          if (ins[i] == '.') dcp=i;
          i++;
        }
        ttl=i;
        ldz=i-1;
        i=0;
    }
    if (ldz<0)
    { ldz = -ldz;
      ch='*';
 }
     while (ch==ins[i] && i<ldz) i++;
 /* leading zeros */
     ch= ins[i];
 /* sign before the number */
     if (ch=='+' && sgnt>0) i++;
     else if (ch=='-' && (sgnt<0 || sgnt>1))
           { i++;
             sig = -1;
}
     while (i<ttl) if (i==-dcp-1 &&  ins[i]=='.') i++;
                   else { /* digits of the number */
                          dig= ins[i]-'0';
                          val=val*10+dig;
                          i++;
                          if (dig<0 || dig>9) errpic=1;
 }
     if (dcp<0) dcp = -dcp;
     val=val*sig;
     i=1;
     while (i++<ttl-dcp+1) val/=10.0;
 /* setting decimal point */
     return(val);
}    /* $END$ */
char *_dec(val,ldz,sgnt,dcp,ttl) /* $START$ */
/*$PARMS: 05,2,1,1,1,1,0 */
double val;
int ldz,sgnt,dcp,ttl;
{ static char string [40];
  char ch = '0',sgn, dig[10];
  int i,ii;
  double corv=0.5;
  dig[0]='0';
dig[1]='1';
dig[2]='2';
dig[3]='3';
dig[5]='5';
dig[6]='6';
  dig[7]='7';
dig[8]='8';
```

```
dig[9]='9';
dig[4]='4';
dig[10]='\0';
   if (ldz<0)
   { ldz = -ldz;
     ch='*';
  }
   else if (ldz>0) ch=' ';
   sgn=ch;
   if (val<0)
       { val= -val;
         if (sgnt<0 || sgnt>1) sgn='-';
       }
   else if (sgnt>0) sgn='+';
   if (dcp<0) ii= -dcp-1;
   else ii=dcp;
   for(i=ii;i<ttl;i++) corv/=10.0;
   val+=corv;
   for(i=1;i<=ii;i++) val/=10.0;
   if (val>=1.0)
       { errpic=1;
         val=0.0;
       }
   for (i=0;i<ttl;i++)
    {   if (i==-dcp-1) string[i]='.';
        else
           { val*=10.0;
             ii=val;
             val-= ii;
             if (sgn=='W') string[i]=dig[ii];
             else if (ii==0 && i<ldz) string[i]=ch;
                    else { if (i>0) string[i-1]=sgn;
                           sgn='W';
                           string[i]=dig[ii];
                         }
           }
    }
   string[ttl]='\0';
   return(string);
}    /* $END$ */
```

APPENDIX 3

MTE source code

1. Lexical Analyzer :

```
%{
/************************** FILE "lex.src" ************************************
    Nov 20th, '87                                                  Mahesh Srinivasan

        This file is the source specification for the lexical analyzer of the
MODEL Timing evaluator. By running the UNIX utility "lex" on this file,
a C file to perform lexical analysis is generated. For more details, refer
to "Lex User Manual" and the report "MODEL TIMING EVALUATOR"
        The first section has variable declarations that will appear literally in
the generated C program and a few "macro" style declarations. The second section
contains rules for splitting the input into tokens and the last section has a
few C functions that will also be literally reproduced in the generated C file.
        The lexical analyzer reads its input from "standard input" and returns an
integer whenever a token is matched. It has been designed to be used in
conjunction with the parser generated by "yacc"

*******************************************************************************/



/* Auxiliary variables */
char s1[30], s2[30], s3[30], s4[30];
char chr,   formats[6][4] = { "%d",
  "%ld","%o","%lo","%x","%lx"};
double num_double;
long num_long;
int num_int, temp;
%}

ws          ([ \t\n])
letter        ([a-zA-Z_])
digit         ([0-9])
positive      ([1-9])
oct           ([0-7])
hex           ([0-9a-fA-F])
exp           ([eE][+-]?(digit)+)
star           (\*)
slash           (\/)
nonstar           ([^\*])
nonstrash      ([^\*\/])

%%
^\#           { /* This rule ignores C preprocessor
  lines, while keeping track
              *   of linenumbers in the source C program.                    */

              char c[100],d[30],e[30];
              int i, j;
                 i = 0;
```

```
                while (( c[i] = input()) != '\n')
                    if (c[i++] == '\\' && (c[i++]
 = input()) == '\n')
                                { linenum++;
                            i -= 2;
                    }
                c[i] = '\0';
                 linenum++;
                            if (sscanf(c,"%d%s%s", &j,d,e) == 2 && d[0] == '"'
 && d[strlen(d)-1] == '"')
                                linenum = j;
            }
(ws)+           {     /* Ignore blanks, tabs and newline
 characters. */
               newline(yytext);

     /* The reserved words of the C language.   */   }
int             return(_INT);
char             return(_CHAR);
float             return(_FLOAT);
double             return(_DOUBLE);
struct             return(_STRUCT);
union             return(_UNION);
long             return(_LONG);
short             return(_SHORT);
unsigned        return(_UNSIGNED);
auto             return(_AUTO);
extern             return(_EXTERN);
register        return(_REGISTER);
typedef             return(_TYPEDEF);
static             return(_STATIC);
goto             return(_GOTO);
return             return(_RETURN);
sizeof             return(_SIZEOF);
break             return(_BREAK);
continue        return(_CONTINUE);
if              return(_IF);
else             return(_ELSE);
for             return(_FOR);
do              return(_DO);
while              return(_WHILE);
switch             return(_SWITCH);
case             return(_CASE);
default             return(_DEFAULT);
entry             return(_ENTRY);
fortran             return(_FORTRAN);
asm             {   return(_ASM);

   /* Identifiers, integer constants, character constants, floating constants,
    * and string constants
 */}
```

```
{letter}({letter}|{digit})*
 {int i;
                        cpstring(strings[i=NEXTSTRING], yytext);
                        yylval.where = i;
                                              return(_IDENTIFIER);
                    }
0{oct}*[lL]?              return(num_scan(2));
{digit}+[lL]?             return(num_scan(0));
0[xX]{hex}*[lL]?          return(num_scan(4));
{digit}+\.{digit}*{exp}?
 |
{digit}*\.{digit}+{exp}?
 |
{digit}+{exp}               { sscanf
(yytext,"%lf",&num_double);
                                              return(_CONS_DOUBLE);                }
\"
([^"\\]|\\.)*\"
              {int i;
                      yytext[yyleng-1] = '\0';
                                              cpstring
(strings[i=NEXTSTRING],yytext+1);
                      yylval.time = (float) (yyleng - 1);
                      yylval.where = i;
                                              return(_CONS_STRING);           }
\'.\'                    { chr = yytext[1];
                                              return(_CONS_CHAR);     }

\'\\[^0-7]\'
    { switch(yytext[2])
                                              {   case 'n' : chr =

 '\n';break;
                                                  case 't' : chr = '\t';break;
                                                  case 'b' : chr = '\b';break;
                                                  case 'r' : chr = '\r';break;
                                                  case 'f' : chr = '\f';break;
                                                 default   : chr = yytext[2];
                                              }
                                              return(_CONS_CHAR);
                                          }
\'\\{oct}{1,3}\'
              { yytext[yyleng-1] = '\0';
                                              sscanf(yytext+2,"%o",&temp);
                                              if (temp == 0)
                                                    chr = '\0';
                                              else chr = (char) temp;
                                              return(_CONS_CHAR);

    /* Comments meant for the Model Timing Evaluator are recognized, but others
     * are ignored.
 */}
{slash}{star}{nonstar}*{star}
```

```
({nonstrash}{nonstar}*{star}+)*{
 slash}     {
                                  char *p,*q;
                          newline(yytext);
                          if ( sscanf(yytext+2,"%29s%d%29s",s1,&num_int,s3) == 3
                                  &&
                                   strcmp(s1,COMMENTAG) == 0
                                  &&
                                   strcmp(s3,"*/") == 0           )
                                                  { yylval.time = (float) num_int;
                                  return(_LOOPRANGE);
                                }
                          else if ( sscanf(yytext+2,"%29s",s1) == 1
                                  &&
                                        strcmp(s1,COMMENTAG) == 0           )
                             { for(p=yytext+2;isspace(*p);p++);
                                                  for(p+=strlen(s1);isspace(*p);p++);
                                                  for(q=s2;isalnum(*p)||*p=='.'||*p=='_';*
(q++)= *(p++));
                                                  *q = '\0';
                                                  cpstring(strings[yylval where =
 NEXTSTRING],s2);
                                return(_FILENAME);
                             }


       /* The operators of the C language          */ }
"\ "              return(_DOT);
"!"              return(_NOT);
"!="              return(_NE);
"%"              return(_MOD);
"%="              return(_MODEQ);
"&"              return(_BITAND);
"&&"              return(_AND);
"&="              return(_BITANDEQ);
"("              return(_LPAREN);
")"              return(_RPAREN);
"*"              return(_MUL);
"*="              return(_MULEQ);
"+"              return(_PLUS);
"++"              return(_INC);
"+="              return(_PLUSEQ);
","              return(_COMMA);
"-"              return(_MINUS);
"--"              return(_DEC);
"-="              return(_MINUSEQ);
"->"              return(_ARROW);
"/"              return(_DIV);
"/="              return(_DIVEQ);
":"              return(_COLON);
";"              return(_SCOLON);
```

```
"<"             return(_LT);
"<<"             return(_LSHIFT);
"<<="             return(_LSEQ);
"<="             return(_LE);
"="             return(_SETEQ);
"=="             return(_EQ);
">"             return(_GT);
">="             return(_GE);
">>"             return(_RSHIFT);
">>="             return(_RSEQ);
"?"             return(_QUESTION);
"["             return(_LSQUARE);
"]"             return(_RSQUARE);
"^"             return(_XOR);
"^="             return(_XOREQ);
"{"             return(_LBRACE);
"|"             return(_BITOR);
"|="             return(_BITOREQ);
"||"             return(_OR);
"}"             return(_RBRACE);
"~"             return(_ONECOMP);
            ;
%%

/* Function that returns the same token for octal, hex and decimal constants. */
num_scan(i)
int i;
{                       if (sscanf(yytext,formats[i],&num_int))
                           { yylval.time = (float) num_int;
                             return(_CONS_INT);
                           }

                        if
(yytext[yyleng-1]=='l'||yytext[yyleng-1]=='L')
                                yytext[yyleng-1] = ' ';

                        sscanf(yytext,formats[i+1],&num_long);
                        yylval.time = (float) num_long;
                        return(_CONS_INT);
}


/* This function is called upon the end of the input stream.            */
yywrap()
{
     return(-1);
}


/* Used to keep track of linenumbers in source program. Called whenever the
    rule for whitespace, "ws" is matched                                  */
```

```
newline(p)
char *p;
{               while (*p != '\0')
                        if (*(p++) == '\n')
                                linenum++;
}


/*                                      ********************                                    */
```

2. Parser :

```
/******************** FILE "yacc.src" ****************************************
    Nov 20th, '87                                                  Mahesh Srinivasan

        This file contains the source specification for the parser of the MODEL
timing evaluator. The Unix utility "yacc" generates, from this file, a C program
to perform the parsing. For more details, refer to the "Yacc User Manual" and
the report "MODEL TIMING EVALUATOR". The parser operates in conjunction with
the lexical analyzer generated by "lex" and also uses numerous auxiliary C
functions found in the file "aux.c"
     In trying to generate code that is as close as possible to the code
generated by the C compiler, care is taken not to generate trivial instructions
eg. type conversions for constants,or constant arithmetic or address calculation
that can be done at compile time. Since the input C program is guaranteed to be
correct, and since the object of interest is the time delay caused by an
instruction rather than the instruction itself, the number of types in the
target machine language have been reduced to two - the same instruction is used
for short integer arithmetic and integer arithmetic since the time delay is
likely to be the same. Lastly, the timing estimation required is a conservative
one - therefore no complicated optimizations are used

*****************************************************************************/

%{
/*   Include definitions    */
#include <stdio.h>
#include <ctype.h>
#include <strings.h>
#include <math.h>

/* The parameter file of the MTE. Contains macro definitions and certain other
    kinds of data that determine the operating parameters of the MTE              */
#include "parameters.h"

/*   This is the type of the value stack of the parser.                    */
typedef struct { float time; short where; type_type type;} stack;
#define YYSTYPE stack

/*   Global variables, declarations and macro definitions.          */
int linenum = 1;
char *malloc();
float eval(), f_proc();
#define MAX(x1,x2)      (((x1)>(x2)) ? (x1) (x2))
#define PMINUS(x)      (((x)>0) ? (x)-1 : 0)

/* File containing the auxiliary functions     */
#include "aux.c"

%}
```

```
     /* Start symbol, token names, and precedences for tokens.                */
%start program

%token _FILENAME _LOOPRANGE _INT _CHAR _FLOAT _DOUBLE _STRUCT _UNION
  _LONG
%token _SHORT _UNSIGNED _AUTO _EXTERN _REGISTER _TYPEDEF _STATIC  GOTO
  _RETURN
%token _BREAK _CONTINUE _IF _ELSE _FOR _DO _WHILE _SWITCH _CASE
  _DEFAULT _ENTRY
%token _FORTRAN _ASM _IDENTIFIER _CONS_DOUBLE _CONS_CHAR _CONS_STRING
  _CONS_INT
%token _LBRACE _RBRACE _SCOLON _ARROW

%left _COMMA
%token _BOGUS
%right _MODEQ _BITANDEQ _MULEQ _PLUSEQ _MINUSEQ _DIVEQ _LSEQ _SETEQ
  _RSEQ _XOREQ _BITOREQ
%left _QUESTION _COLON
%left  OR
%left _AND
%left _BITOR
%left  XOR
%left  BITAND
%left  EQ  NE
%left  LT  GT  LE  GE
%left  LSHIFT  RSHIFT
%left  PLUS  MINUS
%left  MUL  MOD  DIV
%right  NOT  ONECOMP  SIZEOF  INC  DEC
%left  LPAREN  RPAREN  LSQUARE  RSQUARE  ARROW  DOT

%%

  *   The grammar to be used for the parser, and associated actions.  The generated
  *  parser is a bottom up , shift reduce parser.  All nonterminal names are in
  *  lowercase while terminals(tokens) are in upper case                       */

  *   The start production of the grammar.  The parser receives control only after
  *  the "{" in the declaration "main(  ){" has been seen.  It returns control
  *  after successfully parsing the type declarations and body of the main()
  *  function of the input ( program                                           */
program              type_decl_list function_bod,
                  { YYACCEPT; }


  *   This section processes type declarations found at the head of the main()
  *  function.  By means of various flags, declarations of interest ( non integer
  *  scalar or non automatic) are detected and inserted into the appropriate
  *  symbol table with relevant type information                               */
type_decl_list        type_decl_list type_decl_rest  SCOLON
                  { stat_flag   flot_flag   char_flag = 0;
```

```
                    if (s_flag)
                        SPOP;
                    s_flag = 0
                }
            |


type_decl       sc
            | type
            | type_decl sc
            | type_decl type


type            CHAR
                { char_flag++ $$ type = INTGR }
            | SHORT
                { $$ type = INTGR }
            | INT
                { $$ type = INTGR }
            | LONG
                { $$ type = INTGR }
            | UNSIGNED
                { $$ type = INTGR }
            | FLOAT
                { flot_flag++ $$ type = FLOATING }
            | DOUBLE
                { flot_flag++ $$ type = FLOATING }
            | struc_union_spec
                { s_flag = $1 type }


[illegible] type [illegible] determine [illegible] if its definition is present
[illegible] definition [illegible] has been previously seen by the
[illegible] set [illegible] which will cause the variables declared as
[illegible] be inserted in the symbol table
[illegible] spec [illegible] LBRACE
                { $$ type = s_proc[illegible] }
                struc_decl_list RBRACE
                { $$ = $3 }
            | [illegible] IDENTIFIER LBRACE
                                    $$ type = s_proc(GETSTRING
$ where[illegible] )
                struc_decl_list RBRACE
                { $$ = $3 }
            | [illegible] IDENTIFIER
                { $$ type = s_proc(GETSTRING($ where[illegible] ))
            | UNION LBRACE
                { $$ type = s_proc[illegible] }
                struc_decl_list RBRACE
                { $$ = $3 }
            | UNION IDENTIFIER LBRACE
```

```
                                        $$ type = s_proc(GETSTRING($, where)[illegible])

                    struc_decl_list  RBRACE
                        { $$ = $4
            | UNION  IDENTIFIER
                { $$ type = s_proc(GETSTRING($, where)[illegible]

struc_decl_list  struc_decl_tion
            | struc_decl_list struc_decl_tion

struc_decl_tion  type s_declrtr_list  SCOLON
                { stat_flag = flot_flag = bar_flag = 0
                  if (s_flag)
                                  SPOP
                              s_flag = 0
                }

s_declrtr_list  s_declrtr
                { update($1 where) }
            | s_declrtr_list  COMMA s_declrtr
                { update($1 where) }

s_declrtr       declarator
                { $$ where = $1 where }
            | declarator  COLON  CONS_INT
                { $$ where = $1 where }
            |  COLON  CONS_INT
                { $$ where = [illegible] }

                AUTO
            |  STATIC
                { stat_flag++ }
            |  EXTERN
                { stat_flag++ }
            |  REGISTER

 * "update" is called after each declarator has been seen to make an entry in
 * the symbol table for the declarator, if necessary. The argument to this
 * function is the location in the string table of the identifier described
 * by the declarator. Other information about its type and storage class is
 * conveyed by the setting of flags and the variables "n_dim" & "last_dim". */
rest            init_declarator
                { update($1 where), }
            | rest  COMMA init_declarator
                { update($3 where) }
```

```
init declarator        declarator initializer
                { $$ where = $1 where; }
            | declarator
                { $$ where = $1 where; }

 * The basic idea is to store in the symbol table the names of the variables
 * that are declared as either "static" and/or "float" or an array or a pointer
 * to a float or a structure                                                    */
declarator          IDENTIFIER
                    { $$ type = SIMPLE;
                 $$ where = $1 where;
                 if (stat_flag)
                                  { ptr = new();
                      cpstring(ptr->name, GETSTRING($1 where));
                                     insert(ptr, STAT);
                                  }
                               if (flot_flag)
                                  { ptr = new();
                                     cpstring(ptr->name, GETSTRING($1 where));
                                     i_flag++;
                                     ptr->info = FLOATING;
                                  }
                 if (s_flag)
                    { ptr = new();
                      cpstring(ptr->name, GETSTRING($1 where));
                      i_flag++;
                                  }
                            }

            | _LPAREN declarator _RPAREN
                 { $$ type = COMPLEX;
                    i_flag = 0;
                 }

            | _MUL declarator %prec _NOT
                 { $$ type = COMPLEX;
                    $$.where = $2.where;
                    if ($2.type == SIMPLE && flot_flag)
                                       ptr->info = PTRFLT;
                                  if ($2.type == COMPLEX || s_flag)
                                       i_flag = 0;
                          }

            | declarator _LPAREN _RPAREN
                 { $$ type = COMPLEX;
                    $$.where = $1.where;
                    if ($1.type == COMPLEX || s_flag)
                                       i_flag = 0;
```

```
                        }

        | declarator _LSQUARE dimension  _RSQUARE
            { if ($1.type == COMPLEX && n_dim == 0)
                    i_flag = 0;
                        else {$$.type = COMPLEX;
                    n_dim++; i_flag++;
                    last_dim = (int) $3.time;
                  }
              $$.where = $1.where;
                    }
        ;

/* Array dimensions that are given as constant integers are of special interest,
 * hence a separate production rule.                                          */
dimension       : _CONS_INT
            { $$.time = $1.time; }
        |
            { $$.time = 0.0; }
        | exp
            { $$.time = 0.0; }
        ;

/* Not interested in intializations */
initializer     : _SETEQ exp
        | _LBRACE init_list _RBRACE
        | _LBRACE init_list _COMMA _RBRACE
        ;

init_list       : exp %prec _SETEQ
        | init_list _COMMA init_list
        | _LBRACE init_list _RBRACE
        ;

function_body       : _LBRACE type_decl_list stat_list _RBRACE
            { $$ time = $3 time; }
        ;


/*****************************************************************************
 *      The rules for statements  The rule for the "if" statement has deliberately
 *   been made ambiguous  Loop statements should be preceded by a special token,
 *   _LOOPRANGE, inside a comment  The field "time" of the values of
 "stat_list"
 *   and "statement" are used to keep track of the cumulative time delay        */

stat_list
            { $$ time = 0;}

        | stat_list statement
            { $$ time = $1 time + $2 time;
```

```
                if (debug flag)
                    printf("\n");
            }


statement       LBRACE stat list  RBRACE
            { $$ time = $2 time; }

        | exp  SCOLON
            { $$ time = $1 time; }

 * Both paths of the if statement, if present, are processed, appropriate
 * branch instructions are generated, and the longer time delay is the
 * one that is taken into account                                          */
        | IF  LPAREN exp  RPAREN
            { if ($3 where != PSW && $3 where != IMDTE)
                $3 time += eval(CMP,$3 type,IMDTE,$3 where);
              if ($3 where != IMDTE)
                $$ time = eval(BRCOND,DUM,FAIL,DUM);
              else  $$ time = 0;
              IFPUSH

             statement  f rest
                  float t
                      IFPOP
                        t = MAX($5 time+$6 time, $7 time);
                      $$ time = $3 time + t;
                       lock + t = $7 time;


        | prefix  loop
              $$ time = $2 time;

        | BREAK  SCOLON
              $$ time = eval(BR,DUM,DUM,DUM);  }

        | CONTINUE  SCOLON
              $$ time = eval(BR,DUM,DUM,DUM);  }

        | CONTINUE  SCOLON statement
              $$ time = $3 time;  }

        | RETURN  SCOLON
            { $$ time = eval(RET,DUM,DUM,DUM);  }

        | RETURN exp  SCOLON
              $$ time = $2 time + eval(RET,DUM,DUM,DUM);
                        if ($2 where != PSW)
                            $$ time += eval(SCOND,DUM,REG,DUM);
                        else $$ time += eval(MOV,$2 type,$2 where,REG);
```

```
        | SCOLON
            { $$ time = 0;

    * * * statements are treated like nested if statements *
        | SWITCH LPAREN exp RPAREN
            { if ($3 where == PSW)
                { $3 time += eval(SCOND,DUM,REG,DUM);
                  $3 where = REG;
                }
              $3 time += eval(BR,DUM,DUM,DUM);
            }
              LBRACE case list RBRACE
                { $$ time = $3 time + $7 time + eval(BR,DUM,DUM,DUM);



    * * *
            { $$ time = 0;
              clock = $0 time + $ 1 time;
            }
        | ELSE
            { NEGATE
              $0 time += eval(BR,DUM,DUM,DUM);
              clock = $0 time + $ 1 time;
              $$ time = eval(BRCOND,DUM,SUCC,DUM);

          statement
                { $$ time = $2 time + $3 time; }


case list      CASE exp
            { IFPUSH;
              $$ where = $ 3 where;
              $$ time = eval(CMP,INTGR,IMDTE,$$ where) +
                        eval(BRCOND,DUM,SUCC,DUM);
            }
            COLON stat list
                { NEGATE;
                  clock = $3 time + $5 time;
                  $$ time = eval(CMP,INTGR,IMDTE,$3 where) +
                            eval(BRCOND,DUM,FAIL,DUM);
                }
              case list
                { IFPOP;
                  $$ time = MAX($3 time+$5 time, $6 time+$7 time);
                  clock += $$ time = $6 time + $7 time;
```

```
                }

        |   DEFAULT
              { $$ time = eval(BR,DUM,DUM,DUM); }
              COLON stat list
              { $$ time = $2 time + $4 time; }

        |
              { $$ time = 0; }


/* 'looprange' is an integer that tells the MTE the number of times the loop
   will iterate                                                              */
looprange :       LOOPRANGE
              { $$ = $1; }
        |
              { error(0,1); }


/* Appropriate branch instructions are generated and total loop delay is
   calculated using the value of "looprange".  Much of the code below is devoted
   to adjusting "clock", and creating control event "nodes" which are needed for
   generation of the Timing Report, and explained in the file "aux.c"        */
WHILE     WHILE   LPAREN
                { node type *p;
                      $1 time = clock;
                      p = makenode(WHLBEGIN);
                      FORPUSH(p);
                      clock = 0.0;
                }
          exp   RPAREN { node type *p;
                         float t;
                      if ($4 where != PSW)
                      $4 time += eval(CMP,$4 type,IMDTE,$4.where);
                      t = eval(BRCOND,DUM,SUCC,DUM);
                      p = makenode(WHLEXIT);
                      clock = t;
                      p->control lptr = FORTOP;
                      FORPUSH(p);
                      $$ time = eval(BRCOND,DUM,FAIL,DUM);
                }
          statement     { node type *p, *q;
                          $7 time += eval(BR,DUM,DUM,DUM);
                        p = makenode(WHLEND);
                        q = FORPOP;
                        p->control lptr = q;
                        (FORPOP)->control lptr = p;
                        $$ time = ($4 time+$6 time+$7 time)*$0.time
                                  +$4 time +eval(BRCOND,DUM,SUCC,DUM);
                        clock = q->control exit = $1.time + $$.time;
```

```
            }

        | DO          { $1.time = clock;
                      FORPUSH(makenode(DOBEGIN));
                      clock = 0.0;
                    }
          statement _WHILE _LPAREN exp _RPAREN
          { node_type *p;
              if ($6.where != PSW)
                 $6.time += eval(CMP,$6.type,IMDTE,$6.where);
              p = makenode(DOEND);
              (FORTOP)->control.lptr = p;
              p->control.lptr = FORPOP;
              $$.time = ($3.time+$6.time+eval(BRCOND,DUM,SUCC,DUM))*$3.time
                        + eval(BRCOND,DUM,FAIL,DUM);
              clock = p->control.exit = $1.time + $$.time;
          }

        | FOR _LPAREN e_opt
                    { $1.time = clock;
                      FORPUSH(makenode(WHLBEGIN));
                      clock = 0.0;
                    }
            SCOLON e_opt { node_type *p;
                      float t;
                          if ($6.where != NOWHERE && $6.where != PSW)
                              $6.time += eval(CMP,$6.type,IMDTE,$6.where);
                        t = eval(BRCOND,DUM,SUCC,DUM);
                        p = makenode(WHLEXIT);
                        clock += t;
                        p->control.lptr = FORTOP;
                        FORPUSH(NULL);
                        current = &FORTOP;
                        FORPUSH(p);
                        if ($6.where != NOWHERE)
                                $$.time = eval(BRCOND,DUM,FAIL,DUM);
                        else   $$.time = 0.0;
                        $5.time = clock;
                        clock = 0.0;
                    }
          _SCOLON e_opt { current = &FORTOP->control.next;
                      clock = $5.time;
                    }
            _RPAREN statement
            { node_type *p, *q;
                 q = FORPOP;
                 for (*current = FORPOP; *current != NULL; current = &(
(*current)->io.next))
                    (*current)->io.arrival += clock;
                 $12.time += eval(BR,DUM,DUM,DUM);
                 clock += $9.time;
```

```
                    p = makenode(WHLEND);
                    p->control.lptr = q;
                    (FORPOP)->control.lptr = p;
                    $$.time = ($6.time+$7.time+$9.time+$12.time)*$0.time
                              +$6.time + eval(BRCOND,DUM,SUCC,DUM);
                    clock = q->control.exit = $1.time + $$.time;
                    $$.time += $3.time;
                }

        ;


/* Optional control expressions for the "for" loops                    */
e_opt           : exp
                { $$ = $1; }
        |
                { $$.time = 0;
                  $$.where = NOWHERE;
                }
        ;

/* Expressions that are passed as parameters in function calls. Constant
 * integer parameters and string parameters are treated differently because
 * their "values", as defined in the project report, can appear in function
 * timing expressions.                                                  */
expf            : _CONS_INT
                { $$.time = 0;
                  $$.type = INTGR;
                  $$.where = IMDTE;
                  putarg((int) $1.time);
                }
        | exp %prec _NOT
                { $$ = $1;
                  if (ISSTRNG($1.type))
                       putarg(STRNGLEN($1.type));
                }
        ;


/*************************************************************************
      The grammar rules for a C expression. The fields "time", "where" & "type"
 * of the value of an expression are used, respectively, to store the time delay
 * caused by the expression, where the expression is (ie) how it is to be
 * addressed, and the type of the expression (ie) integer, floating, array, etc.
 *                                                                      */

exp             : _IDENTIFIER
                { ident_proc(&$$,$1.where); }

        | constant
```

```
                { $$ = $1; }

            | _LPAREN exp _RPAREN
                { $$ = $1; }

/* Function calls that are preceded by a filename (for critical events) */
            | _FILENAME _IDENTIFIER
                { push();
                    cpstring(CURRENTF.name,GETSTRING($2.where));
                     }
                _LPAREN exp_list_opt _RPAREN
                {tab_elem *p;
                  $$.time = $5.time + eval(FUNC,$1.where,DUM,DUM);
                  $$.where = REG;
                  if ((p = search(CURRENTF.name, SPECIAL)) != NULL)
                    $$.type = FLOATING;
                  else   $$.type = INTGR;
                  pop();
                }

/* Function calls without any filename. */
            |  IDENTIFIER
                { push();
                    cpstring(CURRENTF.name,GETSTRING($1.where));
                     }
                _LPAREN exp_list_opt _RPAREN
                {tab_elem *p;
                  $$.time = $4.time + eval(FUNC,-1,DUM,DUM);
                  $$.where = REG;
                  if ((p = search(CURRENTF.name, SPECIAL)) != NULL)
                    $$.type = FLOATING;
                  else   $$.type = INTGR;
                  pop();
                }

/* Array expressions */
            | exp _LSQUARE exp _RSQUARE
                { $$.time = $1.time + $3.time;
                  if (ISADDR($1.where) && ISCONST($3.where))
                          $$.where = $1.where;
                                else { if (ISADDR($1.where))
                          { $$.time += eval(ADDR,DUM,$1.where,REG);
                                              $1.where = REG;
                                            }
                                        if (!ISCONST($3.where))
                                            { $$.time += eval
(MOV,INTGR,$3.where,REG)
                                                          + eval(MULT,INTGR,IMDTE,REG);
                                              $3.where = REG;
                                            }
                                         $$.time += eval(ADD,INTGR,$3.where,REG);
```

```
                    $$.where = REG;
                }

                        if ($1.type == PTRFLT)
                { $$.time += eval(MOV,INTGR,$1.where,REG);
                  $$.type = FLOATING;
                  $$.where = REG_REL;
                        }
                        else { $$.type = (NUMD($1.type)>0 ?
$1.type : $1.type);
                            if (ISLVALUE($$.type))
                                $$.where = CHWHERE($$.where);
                    }

            }

/* Pointer expression - the MIE does not guarantee support of pointers in the
 * input C program                                                          */
        | MUL exp %prec _NOT
            { $$.time = $2.time;
              $$.type = STAROP($2.type);
              if (ISADDR($2.where))
                                $$.where = CHWHERE($2.where);
              else  switch ($2.where)
                        { case REG  : $$.where = REG_REL; break;
                          default   : $$.where = REG_REL;
                                    $$.time += eval(MOV,INTGR,$2.where,REG);
                        }
            }

/* Structure expression   if structure definition is known, then the result
 * expression's type is known, otherwise it is assumed to be INTEGER.       */
        | exp  DOT  IDENTIFIER
            {tab_elem *p;
                $$.time = $1.time;
                if (ISSTRUC($1.type) &&
                                    (p = getfield(SCODE($1.type),GETSTRING
($1.where))) != NULL)
                    $$.type = p->info;
                else  $$.type = INTGR;
                if (ISLVALUE($$.type))
                    $$.where = CHWHERE($1.where);
                else $$.where = ($1.where == REG ? ADDR_REG:$1.where);
                    }

/* Pointer to a structure  Once again, no error is signalled, but there is a
 * loss of type information about the resulting expression.                 */
        | exp  ARROW _IDENTIFIER
            { $$.time = $1.time;
                $$.type = INTGR;
              $$.where = REG_REL;
```

```
                                    }

* Address operator */
          |  BITAND exp %prec  NOT
               { $$ time = $2 time;
                 $$ type = (ISFLOT($2 type) ? PTRFLT INTGR);
                 $$ time +=  eval(ADDR,DUM,$2 where,REG);
                 $$ where = REG;
               }

* Unary operators */
          |  MINUS exp %prec  NOT
               { $$ time = $2 time;
                 $$ type = $2 type;
                 switch ($2 where)
                   {   case IMDTE    $$ where = IMDTE; break;
                       case PSW       $$ where = REG;
                                   $$ time += eval(SCOND,DUM,REG,DUM)
                                       + eval(NEG,INTGR,REG,REG);
                             break;
                       default        $$ where = REG;
                                   $$ time += eval(NEG,$2 type,$2 where,REG);
                   }
               }

* If "exp" is a relational expression, then no code is generated for the NOT
* operator, since the conditional instruction that is likely to follow can
* be generated with its condition reversed                                    */
          |  NOT exp
               { $$ time = $2 time;
                 $$ type = INTGR;
                 if ($2 where == IMDTE)
                        $$ where = IMDTE;
                 else if ($2 where != PSW)
                       { $$ time += eval(CMP,$2 type,IMDTE,$2 where);
                         $$ where = PSW;
                       }
                        else $$ where = PSW;
               }

          |  INC exp
               { $$ time = $2 time + eval(ADD,INTGR,IMDTE,$2 where);
                                   $$ where = $2 where;
                                   $$ type = $2 type;
                                 }

          |  DEC exp
               { $$ time = $2 time + eval(SUB,INTGR,IMDTE,$2 where);
                                   $$ where = $2 where;
                                   $$ type = $2 type;
                                 }
```

```
        | exp INC
            { $$ time = $1 time + eval(ADD,INTGR,IMDTE,$1 where),
              $$ where = $1 where,
              $$ type = $1 type,
            }

        | exp DEC
            { $$ time = $1 time + eval(SUB,INTGR,IMDTE,$1 where),
              $$ where = $1 where,
              $$ type = $1 type,
            }

        | ONECOMP exp
            { $$ time = $2 time,
                        if ($2 where != IMDTE)
                            $$ where = IMDTE,
                        else { $$ time += eval(COMP,DUM,$2 where,REG),
                               $$ where = REG;
                             }
                        $$ type = INTGR,
                      }

* Compiler function */
        | SIZEOF exp
            { $$ time = 0,
              $$ type = INTGR,
              $$ where = IMDTE,
            }

* Type casts   it is assumed that "type" is a scalar C type                    */
        | LPAREN type cast rest RPAREN exp
            { $$ time = $5 time,
              if ($3 time > 0)
                  $2 type = INTGR,
              $$ type = $2 type,
              if ($2 type != $5 type)
                  {if ($2 type == FLOATING)
                      $$ time += eval(ITOF,DUM,$5 where,REG),
                      else $$ time += eval(FTOI,DUM,$5 where,REG);
                      $$ where = REG;
                  }
              else $$ where = $5 where;
            }

/* Arithmetic operators */
        | exp MUL exp
            { arithop(&$$,&$1,&$3,MULT,1); }
        | exp DIV exp
            { arithop(&$$, &$1, &$3, DIVD,0); }
        | exp MOD exp
```

```
                { arithop(&$$, &$1, &$3, MOD,0); }
            | exp _PLUS exp
                { arithop(&$$, &$1, &$3, ADD,1); }
            | exp MINUS exp
                { arithop(&$$, &$1, &$3, SUB,0); }
            | exp BITAND exp
                { arithop(&$$, &$1, &$3, AND,1); }
            | exp _BITOR exp
                { arithop(&$$, &$1, &$3, OR,1); }
            | exp XOR exp
                { arithop(&$$, &$1, &$3, XOR,1); }
            | exp _LSHIFT exp
                { arithop(&$$, &$1, &$3, ASH, 0); }
            | exp _RSHIFT exp
                { arithop(&$$, &$1, &$3, ASH, 0); }

/* Relational operators */
            | exp _LT exp
                { relop(&$$, &$1, &$3); }
            | exp _LE exp
                { relop(&$$, &$1, &$3); }
            | exp _GT exp
                { relop(&$$, &$1, &$3); }
            | exp _GE exp
                { relop(&$$, &$1, &$3); }
            | exp _EQ exp
                { relop(&$$, &$1, &$3); }
            | exp _NE exp
                { relop(&$$, &$1, &$3); }

/* Logical operators */
            | exp _AND exp
                { logic(&$$, &$1, &$3); }
            | exp _OR exp
                { logic(&$$, &$1, &$3); }

/* Assignment operators */
            | exp _MODEQ exp
                { asgnop(&$$, &$1, &$3, MOD); }
            | exp _MULEQ exp
                { asgnop(&$$, &$1, &$3, MULT); }
            | exp _PLUSEQ exp
                { asgnop(&$$, &$1, &$3, ADD); }
            | exp _MINUSEQ exp
                { asgnop(&$$, &$1, &$3, SUB); }
            | exp _DIVEQ exp
                { asgnop(&$$, &$1, &$3, DIVD); }

/* The different treatment of  the "=" operator is because it is expected to
 * occur frequently, & the code generated is optimized a bit more than usual */
            | exp _SETEQ exp
```

```
                { int sflag, itof flag, ftoi flag;
                  sflag = itof flag = ftoi flag = 0;
                  $$ time = $1 time + $3 time;
                  $$ where = $1 where;
                  $$ type = $1 type;

                  if (ISADDR($3 where))
                    $$ time += eval(ADDR DUM CHWHERE($3 where) $1 where)
                              else {
                              if ($3 where != IMDTE)
                    { if ($3 where == PSW)
                                    sflag++;
                      if ($3 type == FLOATING && !ISFLOT($1 type))
                          ftoi flag++;
                      if ($1 type == FLOATING && !ISFLOT($3 type))
                          itof flag++;
                                }

                                if (sflag && itof flag)
                     $$ time += eval(SCOND DUM REG DUM) +
                                eval(ITOF DUM REG $1 where);
                  else if (sflag)
                              $$ time += eval(SCOND DUM $1 where DUM);
                       else if (itof flag)
                                  $$ time += eval(ITOF DUM $1 where $1 where)
                                          else if (ftoi flag)
                                                    $$ time += eval
(FTOI,DUM,$3 where,$1 where);
                                                         else $$ time += eval
(MOV,$1 type,$3 where,$1 where)
                              } /*end of else*/
                }

            | exp  LSEQ exp
                { asgnop(&$$,&$1,&$3,ASH); }
            | exp  RSEQ exp
                { asgnop(&$$,&$1,&$1,ASH); }
            | exp  XOREQ exp
                { asgnop(&$$, &$1, &$3, XOR); }
            | exp  BITOREQ exp
                { asgnop(&$$, &$1, &$3, OR); }
            | exp  BITANDEQ exp
                { asgnop(&$$, &$1, &$3, AND); }


/* Conditional expression   treated like an "if" statement */
            | exp  QUESTION { if ($1 where != PSW && $1 where != IMDTE)
                        $1 time += eval(MP $1 type,IMDTE $1 where);
                  if ($1 where != IMDTE)
                        $$ time = eval(BRCOND DUM FAIL DUM);
                  else $$ time = 0;
```

```
                IFPUSH;
            }
            exp
                { NEGATE
                    $4 time = eval(BR DUM DUM DUM)
                    clock = $3 time + $4 time
                    $$ time = eval(BRCONC DUM SUCC DUM)
                }
              COLON exp
                { float t
                    IFPOP;
                    t = MAX($3 time+$1 time,$5 time+$7 time)
                    clock += t - $5 time - $7 time
                    if (t == $3 time + $4 time)
                            { $$ = $4
                              $$ time = $1 time + $3 time
                            }
                    else { $$ = $7
                              $$ time = $1 time + $5 time
                            }
                }

        | exp  COMMA exp
            { $$ time = $1 time + $3 time
              $$ type = $3 type
              $$ where = $3 where
            }
        ;
.......................................................................

 * Constants *
constant        CONS INT
            { $$ type = INTGR
              $$ time = 0
                              $$ where = IMDTE
                          }
        |  CONS DOUBLE
            { $$ type = FLOATING
              $$ time = 0;
              $$ where = IMDTE;
            }
        |  CONS STRING
            { $$ type = 2*MAXD + 2 + SCALEF*(int)$1 time
                              $$ where = ADDR ABS;
              $$ time = 0;
            }
        |  CONS CHAR
            { $$ type = INTGR;
              $$ time = 0;
              $$ where = IMDTE
            }
```

```
/* The optional parameter list in a function call */
exp_list_opt        expf ( nextarg(). ) list_rest
                    { $$ time = $1 time + $3 time + arg proc(&$1). }
                |
                    { $$ time = 0. }
                ;

list_rest           list_rest _COMMA expf
                    { $$ time = $1 time + $3 time + arg proc(&$3).
                        nextarg().
                    }
                |
                    { $$ time = 0. }
                ;

/* The latter half of a type cast */
cast_rest           _MUL cast_rest
                    { $$ time = 1 0. }
                | LPAREN cast_rest _RPAREN
                    { $$ time = 1 0. }
                | _LSQUARE dimension _RSQUARE
                    { $$ time = 1 0. }
                |
                    { $$ time = 0 0. }
                ;

/******************** The end of the grammar rules section *****************/
%%

/* The file containing the lexical analyzer */
#include "lex yy c"


/* The MODEL TIMING EVALUATOR's main program.  Opens data files and calls other
 * routines to read the data into memory.  Scans the input program and calls
 * the parser at the appropriate time.  Calls a routine to print the Timing
 * Report, and then exits                                                    */

main(argc, argv)
int argc;
char *argv[];
{ int i;
  FILE *f_data, *i_data, *fopen();

              fprintf(stderr,"\n\n***** MODEL TIMING EVALUATOR
 \n");
              printf("\n\n***** MODEL TIMING EVALUATOR
 \n\n");
```

```
/* Open files */
                if (argc != 3)
                  { fprintf(stderr,"Usage  %s instr_data_file
 func_data_file\n", argv[0]);
                    exit(1);
                  }
                if ((i_data = fopen(argv[1], "r")) == NULL)
                  { fprintf(stderr,"Can't open %s\n",
 argv[1]);
                    exit(1);
                  }
                if ((f_data = fopen(argv[2], "r")) == NULL)
                  { fprintf(stderr,"Can't open %s\n",
 argv[2]);
                    exit(1);
                  }

/* Read data & initialize data structures */
                f_init(f_data);
                i_init(i_data);
                node_init();
                tab_init();
                max_last_d();

/* Look for "main()" and then begin parsing */
                while ((i = yylex()) !=  1 && !(i ==_IDENTIFIER && strcmp
(yytext,"main")==0));
                if (i == -1)
                     error(1,1);
                while (yylex()!=_RPAREN);
                yyparse();

/* Generate timing report */
                wind_up();

                printf("\n\n***** Exiting MODEL timing
 evaluator\n");
                fprintf(stderr,"\n\n***** Exiting MODEL timing
 evaluator\n");

}



/* This function is called by the parser when it detects a syntax error in the
 * input                                                                      */
yyerror()
{
     error(1,1);
}
```

```
 * This function is the action performed [illegible] the [illegible]IDENTIFIER [illegible]
 * expressions. Symbol tables are searched [illegible] determine the expr[illegible] type
 * and storage class.

ident_proc(lhs, strngnum)
YYSTYPE *lhs;
int strngnum;
{   tab_elem *p;
    char a[MAXNAME];

        lhs->time = 0;
            cpstring(a, GETSTRING(strngnum));

        if ((p = search(a, STAT)) [illegible]
                lhs->where = ABS;
        else   lhs->where = REG_RE;

        if ((p = search(a, SPECIAL)) [illegible]
            { lhs->type = p->[illegible]
                if (!ISLVALUE(lhs->type))
                    lhs->where [illegible] ABS [illegible]
            }
        else   lhs->type = INTGR;


 * This function generates [illegible]
float arg_proc(epth)
YYSTYPE *epth;
{
            [illegible]
                [illegible]
            [illegible]
                [illegible]

            [illegible]

 *                          [illegible]
 *                          ..........................
```

3 Auxiliary functions

```
*****************************************************************
        Dec  1st  87                                              [illegible]

    This file contains the [illegible] data [illegible]
MTE parser to perform its [illegible]
first contains macro & type def[illegible]
modes & data types that are [illegible]
section deals with instruct[illegible]
evaluation and the fourth [illegible]
the routines used to bu[illegible]

*****************************************************************

*                SECTION [illegible] Macro def[illegible]

* The string table [illegible]
* table & NEXTSTRING [illegible]

#define TABSIZE 1
char strings[TABSIZE][MAXNAME
int stringptr= 1
#define GETSTRING[illegible]
#define NEXTSTRING [illegible]

* The instruction set [illegible]
#define PBEGIN [illegible]
#define PEND [illegible]
#define RET 2
#define BR 3
#define BRCOND 4
#define SCOND 5
#define ADDR 6

#define MOD [illegible]
#define AND 8
#define OR 9
#define XOR 10
#define COMP 11
#define ASH 12
#define ITOF 13
#define FTOI 14

#define MOV 15
#define ADD 16
#define SUB 17
#define MULT 18
```

```
#define DIVD      19
#define NEG       20
#define CMP       21

#define FUNC      22   /* Not really an [illegible]

/* The various possible values for the [illegible]
 * expression  DUM is used as a dummy parameter [illegible]
 * the instruction has less than 2 operands  NOWHERE [illegible]
 * expressions in cases where they are not present [illegible]
 * expression is a relational one  the [illegible]
 * and the result resides in the bits [illegible]
 * IMDTE correspond to the addressing modes of the target language [illegible]
 * "ADDR" values indicate that an "ADDR" [illegible]
 * the expression, with the source operand addressing mode [illegible]
 * or "REG"_RELative  These values are [illegible]
 * address calculation for arrays and [illegible]
#define DUM       -3
#define NOWHERE   2
#define PSW       1

#define REG       0
#define ABS       1
#define REG_REL   2
#define IMDTE     3

#define ADDR_ABS  4
#define ADDR_REG  5

/* The following macros manipulate the above types
#define     ISADDR(x)   ((x == ADDR_ABS || [illegible] ADDR_REG) [illegible]
#define     ISCONST(x)  ((x == IMDTE || [illegible]
#define     CHWHERE(x)  ((x == ADDR_ABS [illegible] ABS [illegible] REG_REL [illegible]

/* Parameters passed to [illegible] when the [illegible]
 * successful branches have different [illegible]
#define FAIL      0
#define SUCC      1


/* For any variable (or expression) type [illegible] done [illegible]
 * manner    1) MAXD is the maximum number of dimensions [illegible] the
                 program
              2) Pointers to float  type [illegible]
              3) Float variables  Number of array dimensions [illegible]
                 have type = 0
              4) Integer variables  MAXD+1+number of array dimen[illegible]
              5) Character scalars  treated as integer scalars
              6) Character arrays  (2*MAXD+1+num of dimensions [illegible]
                                                                  dimens[illegible]
              7) Structures  2 index [illegible] array of structure def[illegible]
```

```
 * No other type distinction is done. The above coding scheme makes it possible
 * to distinguish between addresses and operands, integer and floating lengths
 * of character strings and different structures. Arrays of structures are
 * handled the same way as scalar structures because a structure by itself is
 * also just an address and cannot be an operand for any operator, the char
 *
 * The macros below manipulate the type of an expression, "x" the argument. *

#define SCALEF      (MAXD+1)
#define PTRFLT      -1
#define INTGR (MAXD+1)
#define FLOATING 0

#define NUMD(x)      (x <= MAXD ? x  (x <= 2*MAXD+1 ? x MAXD 1
(x-2*MAXD-1)%SCALEF))
#define ISLVALUE(x) (x>=PTRFLT && NUMD(x)      0 ? 1 0)
#define ISFLOT(x) (x == 0 ? 1 0)
#define ISINTGR(x)   (ISLVALUE(x) && !ISFLOT(x))
#define ISARRAY(x)   (x >= PTRFLT && x != FLOATING && x != INTGR)
#define ISSTRNG(x)   ( x > 2*MAXD+1 && NUMD(x)      1)
#define STRNGLEN(x) ((x    2*MAXD 1)/SCALEF)
#define      STAROP(x) (x <   MAXD ? FLOATING    INTGR)
#define ISSTRUC(x) ( 2 x >=0 &&  2 x <MAXSTRUCS ? 1 0)
#define SCODE(x) ( 2 x)


/* The list of nodes corresponding to critical events and control events is
 * made up of nodes of "control type" and "io type". They can be distinguished
 * between by the value of "node num" which is, and should always be, the first
 * field in both the nodes. The union "node type" is defined so that both kinds
 * of nodes can be addressed by the same pointer                                  */

typedef struct { short node num,
                      union a  *next,
                      float arrival, exit
                      union a *lptr,          } control type,

typedef struct { short node num,
                      union a *next,
                      float arrival  time,
                      char *name, flname[MAXNAME],
                      short last, ifs[MAXIFNEST], } io type,

typedef union a { control type control,
                       io type io,                 } node type,

node type *makenode(),
```

```
/*****************************************************************************/


/*                SECTION 2    Instruction times      */


/* All the instruction times are stored in a single array, "instr_times"  The
 * macros below help address the correct time in the array, based on the
 * instruction code, its type, and the addressing modes of its operands        */

#define NTIMES 279
float instr_times[NTIMES];

#define NOOP(x) (x < 4)
#define CONDBR(x) (x == 4)
#define CONDSET(x) (x == 5)
#define INSADDR(x) (x == 6)
#define TWOOP(x) (x >= 7 && x <= 14)
#define THREEOP(x) (x >= 15 && x <= 21)

#define INDX(i,t,o1,o2) (NOOP(i)?i (CONDBR(i)?i+o1 (CONDSET(i)?6+o1 (INSADDR
(i)\
?9+(o1-1)*3+o2 (TWOOP(i)?15+(i-7)*12+o1*3+o2 111+(i-15)*24+t*12+o1*3+o2)))))


/* Function that copies the instruction timing data file into the array
 * "instr_times"
 */
i init(fp)
FILE *fp;
{ int i;
            for (i = 0; i< NTIMES; i++)
                if (fscanf(fp,"%f",&instr_times[i]) != 1)
                    error(5,0);

}


/* Instruction names and addressing modes to be printed if "debug_flag" is
 on */
char *ops[] = {"REG","ABS","REG_REL","IMDTE"};
char *instrucs[] = {"PBEGIN","PEND","RET","BR","BRcond","Scond",
 "ADDR",
                        "MOD","AND","OR","XOR","COMP","ASH","ITOF","FTOI",
                        "MOV","ADD","SUB","MULT","DIVD","NEG","CMP" };

/* Clock is incremented every time an instruction or function time delay is
 * evaluated  It is also adjusted in other parts of the program, as explained in
 * the timing report section                                                    */
```

```
float clock = 0.0;

/* Every time an instruction is generated, or a function call encountered,
 * "eval" is called to evaluate its time delay. "instr" is the instruction code,
 * "type" is INTGR or FLOATING, & op1 and op2 are the addressing modes of the
 * operands of the instruction, if any. If instr == FUNC, eval calls another
 * function that returns the time delay of the function, otherwise it references
 * the array "instr_times" and returns the time delay.
 */

float eval(instr, type, op1, op2)
int instr, type, op1, op2;
{ float t;

        if (instr == FUNC)
            t = f_proc(type);
        else { if (instr > BRCOND)
                { op1 = (ISADDR(op1) ? CHWHERE(op1) op1)
                  op2 = (ISADDR(op2) ? CHWHERE(op2) op2);
                }
              if (op1 == op2 && op1 == IMDTE)
                  return(0.0);
              if (op2 == IMDTE)
                { op2 = op1;
                  op1 = IMDTE;
                }

              t = instr_times[IND*(instr-ISFLOT(type)-op1-op2)]
            }

        clock += t;

        if (debug_flag && instr != FUNC)
          { printf("%4d ", linenum);
            printf("%s", instrucs[instr]);
            if (THREEOP(instr))
                printf("%s", (ISFLOT(type) ? f ... ));
            printf("%t");
            if (CONDBR(instr))
                  printf("%s %t",(op1 ? FAIL ? "FAIL" "SUCC"));
            else if (CONDSET(instr))
                        printf("%s", ops[op1]);
                  else if (INSADDR(instr) || TWOOP(instr)||THREEOP(instr))
                              printf("%s %s", ops[op1], ops[op2]);
                        else printf("%t");
            printf("%t%5 if microsecs n%t");
          }

        return(t);

}
```

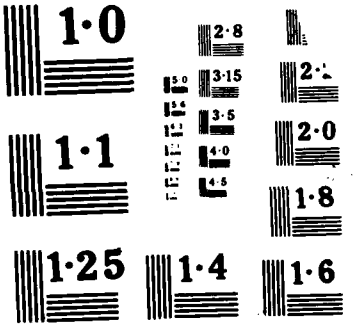
```
/*****************************************************************************

/*                          SECTION 3    Function Times

/* "f_table" is used to store the timing expressions for functions.  Each non
 * blank line of the function timing data file is stored in one string of the
 * "f_table".  n_funcs keeps track of the number of timing expressions in the
 * table.  n_spec_funcs is the number of function names in the array, of
 special
 * functions in the parameter file.  This is not known to the MIE program, but
 * is calculated by "f_init"
 */

char f_table[MAXFUNCS][MAXLEN];
int nfuncs=0;
int n_spec_funcs;

/* Given two timing expressions, f_cmp isolates the function names and returns
 * -1,0, or 1 depending on which name is lexicographically greater.          */
f_cmp(s1, s2)
char *s1, *s2;
{ char n1[MAXNAME], n2[MAXNAME];

                    sscanf(s1, "%19s", n1);
                    sscanf(s2, "%19s", n2);
                    return(strcmp(n1,n2));
}



/* Reads in the function timing data file into "f_table", ignoring blank lines
 * Checks syntax of each line before storing it.  Finally, sorts "f_table" by
 * function name for faster searching later on.                              */
f_init(fileptr)
FILE *fileptr;
{    int i;
     char *p;
                        for (i = 0; i < MAXFUNCS && fgets
(f_table[i],MAXLEN,fileptr) != NULL; i++)
                            { for (p = f_table[i]; *p ==      || *p
  '\t'; p++);
                              if (*p == '\n' || *p == '\0')
                                 { i--;
                                   continue;
                                 }
```

```
                    for (; isalnum(*p) || *p == '_'; p++)
                    if (!valid_exp(p))
                        error(7,0);
                }
            if (i >= MAXFUNCS)
                error(8,0);
            nfuncs = i;
            qsort(f_table, nfuncs, MAXLEN, f_cmp);
            n_spec_funcs = sizeof(special_funcs)/sizeof(char *);
}


/* A small finite state machine to check the syntax of each line of the
 * function data file.  "valid_exp" returns 1 if "s" points to a syntactically
 * valid timing expression, 0 otherwise                                        */

#define BLANK 0
#define DIGIT 1
#define POINT 2
#define ARG   3
#define OPRTR 4
#define OTHER 5

#define ACCEPT(x) ((x == 1 || x == 2 || x == 4) ? 1 : 0)

short control[6][6] = { 0, 1, 3, 3, 5, 5,
                        4, 1, 2, 5, 0, 5,
                        4, 2, 5, 5, 0, 5,
                        5, 2, 5, 5, 5, 5,
                        4, 5, 5, 5, 0, 5,
                        5, 5, 5, 5, 5, 5 };

get_index(c)
char c;
{
        switch (c)
          { case ' ':       return(BLANK);
            case '\t':      return(BLANK);
            case '.':       return(POINT);
            case 'a':       return(ARG);
            case 'A':       return(ARG);
            case '+':       return(OPRTR);
            case '-':       return(OPRTR);
            case '*':       return(OPRTR);
            default:        if (isdigit(c))
                                return(DIGIT);
                            else return(OTHER);
          }
}

valid_exp(s)
```

```
char *s,
{ short state,

                    for (state = 0, *s != ... && *s != ...
 s++)
                            state = contr_[state][get_index(*s)]
                    return(ACCEPT[state])

}


/* Whenever the parser recognizes a function call in the input program, it
 * pushes a "f_stack_elem" on the stack, f_stack. The array, args, [illegible]
 * the "values" of the arguments in the function call, as defined in the
 * project report. A stack structure is needed because of the possibility of
 * nested function calls. Once the whole function call has been parsed, the
 * function "f_proc" is called to evaluate its delay, and then the entry on
 * top of the stack is popped off. Since the "values" of arguments are non-
 * negative integers, the undefined argument values are represented [illegible]

typedef struct { char name[MAXNAME],
                 int c_flag, argptr,
                 int args[MAXARGS], } f_stack_elem

f_stack_elem = f_stack[MAXDEPTH],
int f_ptr = -1, f_flag = 0,

#define CURRENTF = f_stack[f_ptr]


/* Push an entry on to the stack and initialize */
push()
{
        if (++f_ptr > MAXDEPTH - 1)
            error(6,0),
        CURRENTF c_flag = 0,
        CURRENTF argptr = 0,
        f_flag++,
}

/* Pop an entry off the stack */
pop()
{
        if (--f_ptr < -1)
           { printf("ERROR in function stack\n"),
             exit(1),
           }
        f_flag--;
}

/* This function is called after each argument of a function call has been
```

AD-A191 204 A TIMING EVALUATOR FOR C PROGRAMS GENERATED BY THE MODEL SYSTEM(U) RENSSELAER POLYTECHNIC INST TROY NY DEPT OF COMPUTER SCIENCE M SRINIVASAN ET AL. DEC 87 2/2

UNCLASSIFIED RPI-TR-87-29 N00014-86-K-0442 F/G 12/5 NL


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
2·0
1·8
1·4
1·6

```
 * parsed. It fills in a -1 value for the current argument if necessary, and
 * increments argptr.                                                    */
nextarg()
{
        if (CURRENTF.argptr >= 0 && CURRENTF.c_flag == 0)
            CURRENTF.args[CURRENTF.argptr] = -1;
        CURRENTF.argptr++;
        CURRENTF.c_flag = 0;
}

/* Places the value "i" in the current argument of the function entry on top
 * of the stack.                                                         */
putarg(i)
int i;
{
        if (CURRENTF.argptr >= MAXARGS)
            error(13,0);
        CURRENTF.args[CURRENTF.argptr] = i;
        CURRENTF.c_flag++;
}


/* Binary search routine for the sorted "f_table". Returns index in the table
 * of the function name pointed to by "s".                               */
#define NOTFOUND -1

bin_search(s, llimit, ulimit)
char *s;
int llimit, ulimit;
{   int mid, cond;

            if (ulimit < llimit)
                return(NOTFOUND);
            mid = (llimit + ulimit)/2;
            cond = f_cmp(s, f_table[mid]);

            if (cond == 0)
                return(mid);
            if (cond < 0)
                return(bin_search(s, llimit, mid-1));
            if (cond > 0)
                return(bin_search(s, mid+1, ulimit));

}


/* f_calc evaluates the timing expression for the function entry on top of the
 * stack. It obtains the timing expression for the function using "bin_search",
 * performs the necessary argument value substitution, and then calls
 "eval_exp"
 * to evaluate the resulting arithmetic expression.                      */
```

```
float f_calc(fptr)
f_stack_elem *fptr;
{   char exp[MAXLEN-MAXNAME+1], *p, *q, *index();
    int which, i;
    float exp_eval();

            if ((which = bin_search(fptr->name, 0, nfuncs-1)) == NOTFOUND)
                { printf("\'%s\' : ",fptr->name);
                  error(9,1);
                }

            for (p = f_table[which]; isspace(*p); p++); /* Skip leading
 space*/
            for (; !isspace(*p); p++); /* Skip function name */

/* Copy the timing expression into "exp", making substitutions. */
            for (q = exp; *p != '\n' && *p != '\0';)
                if (*p != 'a' && *p != 'A')
                        *(q++) = *(p++);
                else { for (i = 0; isdigit(*++p); i = i*10 + *p - '0');
                        if ( i > fptr->argptr || i < 0 || (i > 0 &&
 fptr->args[i-1] == -1))
                                { printf("\'%s\':
 ",fptr->name);
                                  error(10,1);
                                }
                        if (i > 0)
                                sprintf(q, "%d", fptr->args[i-1]);
                        else   sprintf(q, "%d", fptr->argptr);
                        q = index(q, '\0');
                      }
             *q = '\0';

             return(exp_eval(exp));

}


/* This function receives a character string that is an arithmetic expression
 * with constants & the operators "+", "-", and "*". It recursively calculates
 * the expression and returns the result.                                      */

float exp_eval(s)
char *s;
{    float operand1, operand2,
     char operator[2], *index();

             operand2 = 1;
             while (1)
                    { if (sscanf(s, "%f%1s", &operand1, operator) < 2)
```

```
                    return(operand1*operand2);
                s = index(s,operator[0]) + 1;
                if (s == 0)
                    error(11,0);

                operand2 *= operand1;
                if (*operator == '+')
                    return(operand2 + exp_eval(s));
                if (*operator == '-')
                    return(operand2 - exp_eval(s));
            }
}


/* Finds time delay for the function on top of "f_stack". Prints function
 * instruction if "debug_flag" is on. Searches the array "special_funcs",
 * declared in the parameter file, to check if this function call is a critical
 * event. If so, it creates a node for the event. Finally, returns the time
 * delay.                                                                    */

float f_proc(fname)
int fname;
{ node_type *p;
  int i;
  float t;

        t = f_calc(&CURRENTF);
        if (debug_flag)
            printf("%4d BSR\t%s\t%f
 microsecs\n",linenum,CURRENTF.name,t);

        for (i=0; i < n_spec_funcs; i++)
            if (strcmp(special_funcs[i],CURRENTF.name)==0)
                { p = makenode(0);
                  p->io.time = t;
                  p->io.name = special_funcs[i];
                  if (fname >= 0)
                      cpstring(p->io.flname,GETSTRING(fname));
                  else sprintf(p->io.flname,"on line %id",linenum);
                  break;
                }
        return(t);
}


/*****************************************************************************/
```

```
/*                        SECTION 4 : Timing report                                    */


/* The type declarations of elements in the linked list of nodes used for
 * timing report generation is reproduced here for quick reference :
 *
 * typedef struct { short node_num;   --- identifies the kind of control
node.
 *                        union a   *next;   --- points to next node on the list.
 *                        float arrival, exit;- time of arrival & exit.
 *                        union a *lptr; ------ points to another control node.
 *                                      } control_type;
 *
 * typedef struct { short node_num; --- serial number of I/O node.
 *                        union a *next;----- points to next node on list.
 *                        float arrival, time; -- arrival time & duration time.
 *                        char *name, flname[MAXNAME];-- name of special
function &
 *                                                          filename, if any.
 *                        short last, ifs[MAXIFNEST];--- copy of "if_stack"
 *                                                          and stack pointer.
 *                                                    } io_type;
 *
 * typedef union a { control_type control;
 *                          io_type io;               } node_type;
 *                                                                               */


/* Whenever the parser sees an "if" statement, it assigns it a unique
 * identifying integer, "if_num". When the "then" part of the statement is
 * being parsed, this number resides on top of the "if_stack". When the "else"
 * part of the statement is being parsed, this number is negated and put on the
 * stack. At the end of the if statement, the number is popped off the stack.
 * When a node for a critical event is created, a copy of the current
"if_stack"
 * is made in the node's field, "ifs". This enables the function that generates
 * the timing report to determine if any two nodes lie in mutually exclusive
 * paths (ie), whether they lie within the "then" and "else" parts of the same
 * "if".                                                                         */

short if_stack[MAXIFNEST], if_sp = -1;
short if_num = 0;

#define IFPUSH       (++if_sp < MAXIFNEST ? (if_stack[if_sp] =
 ++if_num): error(2,0))
#define NEGATE (if_stack[if_sp] = - if_stack[if_sp])
#define IFPOP   (if_stack[if_sp--])

/* The "for_stack" is used to store pointers to control nodes associated with a
```

```
 * loop because the value stack of the parser is not capable of holding pointer
 * type data. For each "for" loop & "while" loop in the program, 3 control
 * nodes are created - they are labelled WHLBEGIN, to signify loop beginning,
 * WHLEXIT, to signify the exit point from the loop (at the top), and WHLEND,
 * representing the physical end of the body of the loop. For each loop, to make
 * access faster, the WHLBEGIN node has a pointer to its WHLEND node, WHLEND has
 * a pointer to its WHLEXIT, and WHLEXIT has a pointer to its WHLBEGIN. It is to
 * establish these links that a "for_stack" is needed. In the case of
"do-while"
 * loops, the nodes DOBEGIN & DOEND point to each other.                    */

node_type *for_stack[3*MAXLOOPNEST];
short for_sp = -1;

#define FORPUSH(x) (++for_sp < 3*MAXLOOPNEST?(for_stack[for_sp] =
 x):error(3,0))
#define FORPOP      (for_stack[for_sp--])
#define FORTOP      (for_stack[for_sp])

/* Macro definitions for the labels of control nodes */
#define WHLBEGIN     -1
#define WHLEXIT       -2
#define WHLEND       -3
#define DOBEGIN       -4
#define DOEND       -5

/* Macros to determine the type of the node from the "node_num"    */
#define ISCNTRL(x) (x < 0)
#define ISIO(x)     (x >= 0)

/* "nodenumber" is the unique positive serial number given to I/O nodes as they
 * are added to the list.                                                   */
int nodenumber = 0;
#define NEXTNODE (++nodenumber)


/* The first and last nodes in the list. All other nodes have their storage
 * dynamically allocated.                                                   */

io_type firstnode = {0,NULL,0.0,0.0,"PROGRAM
 BEGIN",{'\0'},-1};
io_type lastnode  = {0,NULL,0.0,0.0,"PROGRAM
 END",{'\0'},-1};

/* "current" always points to the "next" field of the last node in the list.
 * (not "lastnode", but the node most recently added to the list)          */
node_type *(*current) = &firstnode.next;


/* Evaluates time for program beginning */
node_init()
```

```
{
            firstnode.time = eval(PBEGIN,DUM,DUM,DUM);
            if (debug_flag)
                printf("\n");
}


/* The variable "clock" is adjusted when certain control structures are parsed :
 * IF statements : At the end of the "then" statement, the clock is set back to
 * the time it would have been if the "else" path had been chosen  The "else"
 * statement is then parsed and at the end of the "if" statement, the clock is
 * set to the time it would be if the longer path is chosen.
 * LOOPS : After the WHLBEGIN (or DOBEGIN) node is created, clock is set to 0.
 * After the body of the loop has been parsed, clock is set to the time it
 * would be after "looprange" iterations of the loop, & this value is copied
 * into the "exit" field of the WHLEXIT or DOEND node. This field is not used
 * in any other node. Thus, I/O events within loops will have their node arrival
 * times relative to the node arrival time of the WHLBEGIN or DOBEGIN node.
 *
 * The function "makenode" creates a new node of the appropriate type, attaches
 * it to the bottom of the list, sets the arrival time equal to the current
 * clock, and initializes other fields.
 *                                                                          */

node_type *makenode(i)
int i;
{   node_type *p;
    int j;

        if (ISCNTRL(i))
                p = (node_type *) malloc(sizeof(control_type));
        else   p = (node_type *) malloc(sizeof(io_type));

        if (p == NULL)
            error(4,0);
        *current = p;

        if (ISCNTRL(i))
                { p->control.node_num = i;
                  p->control.next = p->control.lptr = NULL;
                  p->control.arrival = clock;
                  p->control.exit = 0.0;
                  current = &p->control.next;
                }
        else   { p->io.node_num = NEXTNODE;
                  p->io.next = NULL;
                  p->io.arrival = clock;
                  for (j = 0, p->io.last = -1; j <= if_sp; j++)
                        p->io.ifs[++p->io.last] = if_stack[j];
                  current = &p->io.next;
                }
```

```
            return(p);

}


node_type *ptrsrc;
int mode;

/* "time_table" points to storage big enough to hold the time delay between
 * every node and every other node in the list. The macros enable addressing
 * of the storage as an array, using integer subscripts or pointers to the
 * nodes on the list.                                                        */

float *time_table;
#define SLOT(row,column) (*(time_table + row*(nodenumber+2) + column))
#define IDSLOT(p1,p2) (SLOT(p1->io.node_num,p2->io.node_num))

/* This function is called after all the parsing is done & it is time to print
 * the timing report. It reports time delays between nodes in tabular form -
 * for an explanation of the report, refer to the project report             */

wind up()
{ int i, j;

/* Attach "lastnode" to bottom of list */
             *current = (node_type *) &lastnode;
             lastnode.time = eval(PEND,DUM,DUM,DUM);
             lastnode.arrival = clock;
             lastnode.node_num = nodenumber + 1;
             debug_flag = 0;

/* Allocate storage to hold time delays */
             time_table = (float *) malloc(sizeof(float)*(nodenumber+2)*
(nodenumber+2));
             if (time_table == NULL)
                  error(4,0);

/* Initialize */
             for (i = 0; i < nodenumber+2; i++)
                 for (j = 0; j < nodenumber+2; j++)
                       SLOT(i,j) = -1.0;

/* Print heading */
             printf("\n\n\n***** TIMING
 REPORT\n");
             printf("\n\tLEGEND\n");
             printf
("\t******\n\nNo.\t%-41s\tTIME
\n---\t%-41s\t----\n","NAME","----");

/* Travel down the list, calculating delays from
```

```
 * current node to every other node on list.        */

            for (ptrsrc = (node_type *) &firstnode, mode = 0;
                        ptrsrc != NULL; ptrsrc = ptrsrc->io.next, mode = 0 )
              if (!ISCNTRL(ptrsrc->control.node_num))
               { node_calc(ptrsrc->io.next,NULL,0.0,0.0);
                 printf("%3d\t%-20s %-20s\t%-0.3f
millisecs\n",ptrsrc->io.node_num,
                                ptrsrc->io.name,ptrsrc->io.flname,ptrsrc->io.time/1E3);
               }

/* Print a nice table */
            print_report();

}


/* "access" determines if the nodes *p1 and *p2 are on mutually exclusive paths
 * or not by looking at their "ifs" fields.                                    */

#define YES 1
#define NO  0

access(p1,p2)
node_type *p1, *p2;
{   int i, j;

                for (i = 0; i <= p1->io.last; i++)
                    for (j = 0; j<= p2->io.last; j++)
                        if (p1->io.ifs[i] == -p2->io.ifs[j])
                            return(NO);

                return(YES);
}



float maxtime = -1.0;
int width;

/* "t_calc" calculates time delay between the nodes *p1 and *p2 using the time
 * offsets o1 and o2. If mode is -1, and a delay has been previously stored,
 * then a new delay is not calculated. Such situations occur in the case of
 * nodes within nested loops. The time offsets are also required when one or
 * both nodes is within a loop(s).                                             */

t_calc(p1,p2,o1,o2)
node_type *p1, *p2;
float o1, o2;
{    float t;
```

```
                t = p2->io.arrival + o2 - p1->io.arrival - o1;

                if ( (mode == -1 && IDSLOT(p1,p2) == -1)
                                 ||
                     (mode == 0 && access(p1,p2) == YES) )

                    {   IDSLOT(p1,p2) = t;
                        if (t > maxtime)
                            maxtime = t;
                    }

}



/* A recursive function that calculates and stores time delays between the node
 * "*ptrsrc" and the nodes on the list starting from "*from" to "*to"  The 2
 * offsets apply to the source node & destination node. A complicated function.
 *                                                                              */

node_calc(from,to,soffset,doffset)
node_type *from, *to;
float soffset, doffset;
{    node_type *p;

        for (p = from; p != to; p = p->io.next)
            if (!ISCNTRL(p->io.node_num))
                    t_calc(ptrsrc,p,soffset,doffset);
            else   switch (p->control.node_num)
                    { case WHLBEGIN : ;
                      case DOBEGIN   :node_calc(p->control.next,p->control.lptr,
                                                soffset, doffset+p->control.arrival);
                                      p = p->control.lptr;
                                      break;

                      case WHLEND :
 mode = -1;
 node_calc(p->control.lptr->control.lptr->control.next,p,
                                soffset - p->control.arrival,doffset);
 mode = 0;
 soffset += p->control.lptr->control.exit - p->control.lptr->control.arrival -
            (ptrsrc->io.arrival + soffset > p->control.lptr->control.arrival) *
                                                                    p->control.arrival;
 break;

                      case DOEND :
 mode = -1;
 node_calc(p->control.lptr->control.next, p,
            soffset - p->control.arrival - eval(BRCOND,DUM,SUCC,DUM), doffset);
 mode = 0;
```

```
 soffset += p->control.exit - p->control.arrival - eval(BRCOND,DUM,FAIL,DUM);
                    break;

                      } /* end of switch */

} /* end of node_calc */


/* Draws a dotted line */
line(i,k)
int i,k;
{    int j;
               printf(" ");
               for (j=0; j < i; j++)
                  if (k && (j == 0 || (j-5)%width == 0))
                       printf("|");
                  else printf("-");
               printf("\n");
}


/* Prints time delays in a neat table */

#define MIN(x1,x2) (((x1) < (x2)) ? (x1):(x2))
char fm1[10], fm3[10], fm2[20];
double log10();

print_report()
{ int n, row, col, leng, i;
   int ndigs,lead,trail;

                    printf("\n\n\tAll times in
 MILLISECONDS\n");
                    printf("All times FROM beginning of one event
 %s\n\n",
                                                        "TO beginning of next event");

                    ndigs = MAX(3,(int) log10((double) (maxtime/1E3)) + 1);
                    width = ndigs+7;
                    lead = (width - 1)/2;
                    trail = width - lead - 2;
                    sprintf(fm1,"%%%1ds%%2d%%%1ds|",lead-1,trail);
                    sprintf(fm3,"%%%1ds**%%%1ds|",lead-1,trail);
                    sprintf(fm2," %%%1d.3f |",ndigs+4);

                    for (n = 1; (n-1)*NCOLS < nodenumber+2; n++)
                       { leng = 6+ width*(MIN(n*NCOLS-1,nodenumber+1) (n-1)*NCOLS +1);
                         line(leng,0);
                         printf("|      |");
                         for (i=0; i< (leng-14)/2; printf(" "),i++);
```

```
                        printf("<- TO ->");
                        for (i=0; i< (leng-16)/2; printf(" "),i++);
                        printf("|\n");
                        printf(" |From|");
                        for (row = -1; row < nodenumber+2; row++)
                            { if (row != -1)
                                 printf(" |%3d |",row);
                              for (col = (n-1)*NCOLS;
                                        col <= n*NCOLS - 1 && col < nodenumber+2;
 col++)
                                  if (row == -1)
                                        printf(fm1," ",col," ");
                                  else if (SLOT(row,col) >= 0)
                                             printf(fm2,SLOT(row,col)/1E3);
                                        else printf(fm3," "," ");
                              printf("\n");
                              if (row == nodenumber+1)
                                    line(leng,0);
                              else line(leng,1);
                            }
                        printf("\n\n");
                      }

                   printf("Total Execution time = %-0.3f millisecs\n",SLOT
(0,nodenumber+1)/1E3);

}



/*****************************************************************************/



/*                              SECTION 5 : Symbol tables & Code generation          */

/* There are 2 symbol tables - tables[0] is used to store variable names
 whose
 * storage class is "static". For this case, the field "info" of "tab_elem" is
 * not used, but a different type has not been defined for the sake of easy
 * implementation. tables[1] is used to store variable names whose type
 is not
 * INTGR. The various types and their mapping to short integer are discussed in
 * section 1. The field "info" carries the type information, & is copied as it
 * is into the "type" field of the value of an identifier in the parser.
 * The symbol tables are implemented by open hashing, with a primitive hashing
 * function.                                                                    */
```

```
#define N_TABLS 2
#define THERE 0
#define NOTHERE 1
#define ERROR -1

typedef struct x { char name[MAXNAME];
                   type_type info;
                   struct x *next;        } tab_elem;

tab_elem *tables[2][N_BUCKS];
tab_elem *search(), *new();


/* Initialize "buckets" to NULL */
tab_init()
{ int i, j;

            for (i = 0; i < N_TABLS; i++)
                for (j = 0; j < N_BUCKS; j++)
                    tables[i][j] = NULL;
}


/* Make sure identifier names are truncated to MAXNAME characters */
cpstring(s1,s2)
char *s1, *s2;
{
            strncpy(s1,s2,MAXNAME);
            s1[MAXNAME-1] = '\0';
}



/* Hashing function */
hash(name)
char *name;
{ int tot;
  char *p;
                for (tot = 0, p = name; *p != '\0' && p - name <
 MAXNAME-1; p++)
                    tot += *p;
                return(tot % N_BUCKS);
}


/* Insert *ptr into tables["table"] */
insert(ptr, table)
tab_elem *ptr;
int table;
{tab_elem *p;
```

```
 int i;
                        if (ptr == NULL)
                             return(ERROR);
                        if ((p = search(ptr->name, table)) != NULL)
                               return(THERE);
                        i = hash(ptr->name);
                        ptr->next = tables[table][i];
                        tables[table][i] = ptr;
                        return(NOTHERE);
}


/* Search for *name in tables[table] */
tab_elem *search(name, table)
char *name;
int table;
{tab_elem *p;
                        if (name == NULL)
                            return(NULL);
                        for (p = tables[table][hash(name)]; p != NULL; p
 = p->next)
                            if (strcmp(p->name, name) == 0)
                                 return(p);

                        return(NULL);
}


/* Return pointer to a new "tab_elem" */
tab_elem *new()
{ tab_elem *p;
                       p = (tab_elem *) malloc(sizeof(tab_elem));
                       if (p != NULL)
                            return(p);

                       error(4,0);
}


/* Because of the way types are encoded for character arrays, there is a limit
 * on the size of the last dimension of a character array. This function
 * computes the limit, and if any arrays in the input program are found to
 * exceed this limit, an error message is generated, asking the user to change
 * the definition of "type_type" to a bigger integral data type.
 */

int maxId;
max_last_d()
{
              maxId = (( (int) (pow((double) 2,(double) (sizeof(type_type)*8-1)))
                                                                    -3*MAXD-1)/SCALEF);
```

```
}


/* "s_stack" is used to handle nested declarations of structures  This is due
 * to the fact that outer level structure declarators have to be inserted in
 * the symbol table while inner level declarators that are themselves fields
 * in other structures have to be inserted in the appropriate list in the array
 * of structure definitions. An empty stack signals that the declarator being
 * processed should go to the symbol table, otherwise it should go to the list
 * whose index in the structure definition array is given by the top stack
 * number.                                                                    */

int s_stack[MAXSSTACK], s_num= -1;

/* "structures" is the array that stores structure definitions. Each element of
 * the array corresponds to the definition of the structure whose tag (if any)
 * is in the field "name" - it is a header of a linked list of "tab_elem"s and
 * list contains all the fields of the structure whose type is not INTGR. The
 * criteria for inclusion of fields in the list is the same as those for the
 * symbol table; so is the manner of coding of type information. The only
 * difference being the place where "tab_elem" is put in.                     */

typedef struct { char name[MAXNAME];
                 tab_elem *fields;    } header;
header structures[MAXSTRUCS];
int last_struc= -1;

/* Macros that operate on s_stack */
#define SPUSH(x) (++s_num < MAXSSTACK ? (s_stack[s_num] = x) : error
(12,0))
#define SPOP       (s_stack[s_num--])
#define SFIRST     (s_num >-1 ? s_stack[s_num] : -1)
#define SSECOND    (s_num-1 > -1 ? s_stack[s_num-1] : -1)
#define EMPTYSTACK (s_num < 0)


/* Links *p to the list whose header is structures["sno"] */
link(p,sno)
tab_elem *p;
int sno;
{
                p->next = structures[sno].fields;
                structures[sno].fields = p;
}


/* Searches for a field "*name" in the list structures["sno"].fields */
tab_elem *getfield(sno,name)
int sno;
char *name;
```

```
{   tab_elem *p;

                for (p = structures[sno].fields; p != NULL; p = p->next)
                    if (strcmp(p->name, name) == 0)
                        return(p);
                return(NULL);
}


/* Prepares a header for a new structure "*name" definition */
getstruc(name)
char *name;
{
                if (++last_struc >= MAXSTRUCS)
                     error(14,0);

                cpstring(structures[last_struc].name, name);
                structures[last_struc].fields = NULL;
                return(last_struc);
}


/* Tries to find an existing header for structure *name */
findstruc(name)
char *name;
{   int i;
                for (i = 0; i <= last_struc; i++)
                    if (strcmp(structures[i].name, name) == 0)
                        return(i);
                return(-1);
}


/* Puts new structure's index on s_stack.If code is 1 and findstruc() fails,
 * it is implied that the structure *name's definition is not known. The wise
 * course of action is to act like the type declaration seen was "int".      */
s_proc(name,code)
char *name;
int code;
{   int i;
                if (code == 0)
                        i = getstruc(name);
                else    i = findstruc(name);
                if (i == -1)
                     return(0);
                SPUSH(i);
                return(1);
}
```

```
/* The various flags and counters are set while parsing each type declaration.
 * Based on these global variables, "update" decides whether and where to store
 * the declarator and its type information.                                    */

#define SIMPLE  0
#define COMPLEX  1
#define SPECIAL  0
#define STAT      1

int stat_flag=0, char_flag=0, flot_flag=0, i_flag=0, s_flag=0;
int n_dim=0, last_dim=0;
tab_elem *ptr= NULL;

update(s)
int s;
{   int i;

                if (n_dim > MAXD)
                     error(15,0);

                if (last_dim > maxid)
                     error(16,0);

                if (i_flag && n_dim >0)
                     if (flot_flag)
                             ptr->info = n_dim;
                     else {ptr = new();
                             cpstring(ptr->name,GETSTRING(s));
                             if (char_flag)
                                     ptr->info = 2*MAXD + 1 + n_dim + last_dim*SCALEF;
                             else   ptr->info = INTGR + n_dim;
                            }

                 if (i_flag)
                       if (!s_flag)
                          if (EMPTYSTACK)
                                insert(ptr, SPECIAL);
                          else link(ptr, SFIRST);
                       else { ptr->info = SCODE(SFIRST);
                                if ((i = SSECOND) != -1)
                                       link(ptr,i);
                                else insert(ptr, SPECIAL);
                              }

                 n_dim = last_dim = i_flag = 0;
                 ptr = NULL;
}
```

```
/*              ******* Code Generation ********       */


/* This section consists of functions used by the parser to assist in code
 * generation for expressions.                                              */

#define ONE 1
#define TWO 2
#define ARITHMETIC 0
#define RELATIONAL 1

#define RESTYPE(x1,x2) (x1 == x2 ? x1 : FLOATING)
#define ADRI(x1,x2) (ISADDR(x1) && x2 == IMDTE ? 1:0)



/* This function is called when it has to be ensured that the operand of an
 * expression resides in memory (or a register).                            */
get_lvalue(oprnd)
YYSTYPE *oprnd;
{
                    if (oprnd->where == PSW)
                       { oprnd->time += eval(SCOND,DUM,REG,DUM);
                         oprnd->where = REG;
                       }

                    if (ISADDR(oprnd->where))
                       { oprnd->time += eval(ADDR,DUM,oprnd->where,REG);
                         oprnd->where = REG;
                       }
}


/* This function makes sure that the operands of an instruction are in memory &
 * are of the same type. Further, for arithmetic instructions, it checks if
 * atleast one operand is in a register (for storing the result) and if the
 * instruction is not commutative, it ensures that that oprnd1 (the destination)
 * is in a register.                                                        */

get_ops(oprnd1, oprnd2,iscom, instr)
YYSTYPE *oprnd1, *oprnd2;
int iscom, instr;
{
                    get_lvalue(oprnd1);
                    get_lvalue(oprnd2);

                    adjst_types(oprndi,oprnd2);

                    if (instr == RELATIONAL)
                         return;
```

```
                if (oprnd1->where == REG)
                    return(ONE);
                if (oprnd2->where == REG && iscom)
                    return(TWO);

                oprnd1->time += eval(MOV,oprnd1->type,oprnd1->where,REG);
                oprnd1->where = REG;
                return(ONE);
}



/* To convert the operands to the same type. ARRAY is considered a separate
 * type because expressions involving an array and an integer usually imply
 * some kind of address arithmetic where the integer will have to be scaled
 * up by the dimension of the array.                                        */

adjst_types(oprnd1, oprnd2)
YYSTYPE *oprnd1, *oprnd2;
{

        if ((ISARRAY(oprnd1->type) && ISARRAY(oprnd2->type)) ||
            (ISINTGR(oprnd1->type) && ISINTGR(oprnd2->type)) ||
            (ISFLOT(oprnd1->type)  && ISFLOT(oprnd2->type) )       )
                return;

        arrayint(oprnd1, oprnd2);
        arrayint(oprnd2, oprnd1);
        flotint(oprnd1, oprnd2);
        flotint(oprnd2, oprnd1);

}


/* If op1 is an array, & op2 is an integer, code is generated to scale up the
 * integer                                                                  */
arrayint(op1,op2)
YYSTYPE *op1, *op2;
{
        if (ISARRAY(op1->type) && ISINTGR(op2->type))
           { if (op2->where != IMDTE)
                { op2->time += eval(MOV,INTGR,op2->where,REG) +
                               eval(MULT,INTGR,IMDTE,REG);
                  op2->where = REG;
                }
             op2->type = op1->type;
           }
}


/* If op1 is FLOATING and op2 is INTGR, the integer is converted to floating
```

```
 * point.                                                                    */
flotint(op1,op2)
YYSTYPE *op1, *op2;
{
          if (ISFLOT(op1->type) && ISINTGR(op2->type))
             { if (op2->where != IMDTE)
                  { op2->time += eval(ITOF,DUM,op2->where,REG);
                    op2->where = REG;
                  }
               op2->type = FLOATING;
             }
}



/* Code for arithmetic instructions.  The result is always left in a
 * register.                                                                 */

arithop(result, oprnd1, oprnd2, oprtr,iscom)
YYSTYPE *result, *oprnd1, *oprnd2;
int oprtr, iscom;
{
          result->time = 0;
          if (oprnd1->where == IMDTE && oprnd2->where == IMDTE)
             { result->where = IMDTE;
               result->type = RESTYPE(oprnd1->type, oprnd2->type);
               return;
             }

         if (ADRI(oprnd1->where,oprnd2->where)||ADRI(oprnd2->where,oprnd1->where))
             { result->where = (ISADDR
(oprnd1->where)?oprnd1->where:oprnd2->where);
               result->type = (ISADDR(oprnd1->where)?oprnd1->type:oprnd2->type);
               return;
             }

          if (get_ops(oprnd1, oprnd2, iscom,ARITHMETIC) == ONE)
                result->time += eval(oprtr,oprnd1->type,oprnd2->where,REG);
          else result->time += eval(oprtr,oprnd2->type,oprnd1->where,REG);

          result->time += oprnd1->time + oprnd2->time;
          result->where = REG;
          result->type = oprnd1->type;
}



/* Code for relational operators  The result is always left in the PSW      */
relop(result, oprnd1, oprnd2)
YYSTYPE *oprnd1, *oprnd2, *result;
```

```
{
                result->time = 0;
                if (oprnd1->where == IMDTE && oprnd2->where == IMDTE)
                   { result->where = IMDTE;
                     result->type = INTGR;
                     return;
                   }

                get_ops(oprnd1, oprnd2, 1, RELATIONAL);
                result->time += oprnd1->time + oprnd2->time +
                                        eval(CMP,oprnd1->type,oprnd1->where,oprnd2->where);
                result->where = PSW;
                result->type = INTGR;
}



/* Code for assignment operators other than "="  The result is always left
 * wherever oprnd1 resides.                                                    */

asgnop(result, oprnd1, oprnd2, oprtr)
YYSTYPE *oprnd1, *oprnd2, *result;
int oprtr;
{
                if (oprnd2->where == PSW)
                   { oprnd2->time += eval(SCOND,DUM,REG,DUM);
                     oprnd2->where = REG;
                   }

                if (oprnd2->where != IMDTE)
                   { if (oprnd2->type == FLOATING && !ISFLOT(oprnd1->type))
                        { oprnd2->time += eval(FTOI,DUM,oprnd2->where,REG);
                          oprnd2->type = INTGR;
                          oprnd2->where = REG;
                        }
                     else if (oprnd1->type == FLOATING && !ISFLOT(oprnd2->type))
                                { oprnd2->time += eval
(ITOF,DUM,oprnd2->where,REG);
                                  oprnd2->type = FLOATING;
                                  oprnd2->where = REG;
                                }
                   }

                result->time = oprnd1->time + oprnd2->time
                                        + eval
(oprtr,oprnd1->type,oprnd2->where,oprnd1->where);
                result->where = oprnd1->where;
                result->type = oprnd1->type;

}
```

```
/* Logical operators - "&&" and "||"  It is always assumed that both op1 and
 * op2 have to be evaluated - for a pessimistic time estimate  The result
 * is left in the PSW                                                    */

logic(result, op1, op2)
YYSTYPE *result, *op1, *op2;
{
                result->time = op1->time + op2->time;
                if (op1->where == IMDTE && op2->where == IMDTE)
                   { result->where = IMDTE;
                     result->type = INTGR;
                     return;
                   }

               if (op1->where != PSW)
                     result->time += eval(CMP,op1->type,IMDTE,op1->where);
               if (op2->where != PSW)
                     result->time += eval(CMP,op2->type,IMDTE,op2->where);

               result->time += eval(BRCOND,DUM,FAIL,DUM);
               result->where = PSW;
               result->type = INTGR;
}

/*************************************************************************/



/* Error messages generated by the MTE - all errors are fatal and cause the
 * MTE to exit with return code of 1. All error messages are printed on
 * standard error as well as standard output.                            */

char *error_msgs[] = {
"Missing loop range",
"Bad C program",
"Internal error: increase definition of \'MAXIFNEST\' in
 parameter file",
"Internal error: increase definition of \'MAXLOOPNEST\' in
 parameter file",
"Insufficient memory for dynamic allocation",
"Bad instruction data file",
"Internal error: increase definition of \'MAXDEPTH\' in
 parameter file",
"Syntax error in function data file",
"Internal error: increase definition of \'MAXFUNCS\' in
 parameter file",
"No timing data",
"Insufficient argument data to evaluate function timing expression",
"Internal error in function timing evaluation: PANIC!!!",
```

```
"Internal error: increase definition of \'MAXSSTACK\' in
 parameter file",
"Internal error: increase definition of \'MAXARGS\' in
 parameter file",
"Internal error: increase definition of \'MAXSTRUCS\' in
 parameter file",
"Internal error: increase efinition of \'MAXD\' in parameter
 file",
"Internal error: change definition of \'type_type\' to
 bigger data type in parameter file"
) ;



error(i,j)
int i,j;
{
          if (j == 0)
               fprintf(stderr,"ERROR\n");
          fprintf(stderr,"%s\n", error_msgs[i]);
          if (j == 1)
              fprintf(stderr,"ERROR on line %ld of C program\n",
 linenum);
          fprintf(stderr,"\n***** Exiting MODEL timing
 evaluator\n\n");
          printf("\n***** Exiting MODEL timing evaluator due to
 error\n\n");
          exit(1);
}



/*                              ************************                              */
```

END

DATE

FILMED

5-88

DTIC